University Politehnica of Bucharest
Faculty of Automatic Control and Computers
Computer Science and Engineering Department

Diploma Thesis

Task scheduling in Wireless Sensor Networks

by

Voinescu Andrei

Supervisor: Prof. Dr. Ing. Nicolae Țăpuș
Supervisor: As. Drd. Ing. Dan Ștefan Tudose

Bucharest, July 2009

Contents

Abstract

Wireless Sensor Networks are composed of various electronic devices with wireless communication capabilities denoted as nodes. They are used to build “smart applications” that benefit from large amounts of sensing data made available by these networks.

In the context of Wireless Sensor and Actuator Networks (WS&ANs), obtaining sensor data is relatively easy with a single-sink, single-hop approach, sensor data is transmitted to the gateway, which passes the data on to another network (e.g. the Internet).

Heteregenous wireless sensor networks are a different matter. Nodes cannot be treated the same way as they do not have the same capabilities, or types of sensors, actuators, energy sources and so on. Therefore a different approach is needed, one that takes into account the differences between nodes as well as the purpose of the network.

A task-based system is more appropiate in the sense that a task exists for collecting data (e.g. a temperature acquisition task), for processing data or for activating actuators. Such a task-based network requires however an entity that assigns specific tasks to appropriate nodes, assuring minimal energy loss for each task that is run by the network, maximizing network lifetime, as well as taking into consideration several other constraints, such as data dependencies, minimizing radio communication, etc.

The main contribution of this study is treating this problem as a partitioning problem of the tasks into groups, each group representing one of the wireless nodes. We prove that the algorithm we propose is also very versatile and can include all of the constraints needed for energy-efficient wireless applications.

Keywords Wireless Sensor Networks, task, scheduling, graph cuts

Acknowledgements

First I would like to express my gratitude to my advisor prof. dr. Nicolae Țăpuș for his support in the SENSEI project. This study has forked from work on that project and i am grateful that i could be part of the team.

I would also like to give special thanks to my co-advisor, Dan Tudose, for helping me overcome hardware difficulties, for helping me correct this work and for great moral support. I am also very grateful to Emil Slușanschi for the sum of great feedback i've received from him for the early drafts and to Răzvan Tătăroiu for bouncing off ideas with me. This study would not have been possible withouth them.

This research is partially supported by the European Union through the FP7 Programme (of which the SENSEI project is part of).

Last but not least I would like to say "Thank you" to my parents and girlfriend, who have shown understanding and have given me the moral support needed.

Chapter 1

Introduction

"A network of possibly low-size and low-complex devices denoted as nodes that can sense the environment and communicate the information gathered from the monitored field (e.g. an area or volume) through wireless links; the data is forwarded, possibly via multiple hops relaying, to a sink (sometimes denoted as controller or monitor) that can use it locally, or is connected to other networks (e.g. the Internet) through a gateway. The nodes can be stationary or moving. They can be aware of their location or not. They can be homogenous or not." as quoted from [CK03].

The domain of Wireless Sensor and Actuator Networks (WS&ANs) is an emergent multi-disciplinary field, because its applications intertwine with a lot of other domains, such as medicine, biology, meteorology, different types of engineering. Whether it's for an irrigation system, package monitor, pulse monitor or weather station, wireless sensor networks have gained a serious footing in applications that need advanced sensor systems.

They were first designed for military applications, where one can imagine deployment is infinitely simpler without wires, one needs only to position the sensors and turn them on, while redeployment can mean just moving them around. As things evolved, now we can expect to integrate this technology with just about any relevant field where gathering more data helps the system make more intelligent decisions.

Taking into consideration data flow, there are several types of WSNs: single-hop/single-sink, that have a star-like topology and a single gateway, multi-hop / multi-sink, single-hop / multi-sink and multi-hop / single-sink. Single sink refers to the direction in which sensor data flows, that is from source (sensor nodes) to a data sink (a gateway connecting the wireless sensor network to a different type of network, which can be the Internet). Throughout this study we

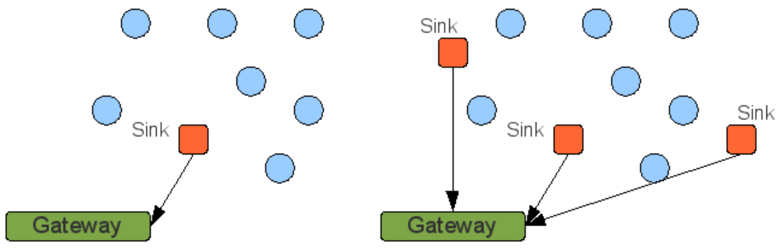


Figure 1.1: *A single-sink wireless sensor network (left), and multi-sink version (right). The gateway is connected to an external network, possibly the Internet*

will consider single-sink networks, single-hop or multi-hop. This type of network can be extended to include WS&AN networks, by including actuator capabilities on the nodes. A single sink for sensor data remains, but now data can flow in different directions, the nodes that have actuators can be commanded either by themselves, or by a neighbouring node, from the services running on the gateway or from the outside (via the gateway). We will use WSN and WS&AN terms intermittently in the study, as some properties are easily generalized.

There are several features a WSN can have:

- *Self-organization* - Refers to the emergence of global behaviour of a network from high-level rules on individual rules on the nodes. As WSNs become more and more popular, the desire would be to minimize human intervention, to have the network organize itself. Also, a WSN with self-organization will be adaptable (it will be able to react to changes in the environment), robustness (it will have the capability to repair damage to itself - self-healing), scalable (the WSN will work just as well with a lot more nodes). [PB05]

- *Self-healing* - Gives the network the possibility to recover from failure of individual nodes, to continue to offer the same services, as well as maintain network integrity by rerouting around failed nodes.

- *Energy efficiency* - Dictates every aspect of a WSN. Consumption has to be kept low in a WSN in order to keep network lifetime to a maximum, either by minimizing radio communication or data processing or balacing it with energy harvesting.

- *Connectivity* - A network has to have to have sufficient connection and alternate routes between nodes, such that each node can communicate with every other node even if one or more nodes have failed.

- *Low-Complexity* - Result of the fact that nodes must be small and minimal processing is needed. In WSNs, not a lot of processing power in needed, as tasks that are required of the nodes are rather uncomplicated. Furthermore, lower complexity may mean smaller size, which is a desired feature.

- *Low-Cost* - In order to be a viable alternative to wired sensor systems, WSNs must offer low maintenance cost as well as low node cost. The cheaper the nodes, the more can be integrated into a WSN with the same price, the greater the advantage over wired networks. Adding a new node to a WSN costs exactly as much as one node, no secondary expenses are needed.

- *Size of nodes* - Has to kept small in order to be able to integrate them well into the environment they will interact with. The ultimate goal here is to build "smart dust", top-of-the-finger (or even smaller) that can act as basic nodes in a WSN.

Chapter 2

Related Work

Related work for task scheduling is generally found under "task mapping" or "task allocation". In recent times many articles discuss this topic, as it is fundamentally different to actively researched topics that mark a certain resemblance, such as Task Scheduling in high performance computing systems. WSNs have unique requirements in respect to network lifetime, availability, reliability that make research previously done not applicable. The objective of these articles is to find schedules with balanced energy consumptions for tasks. These schedules will allow a higher processing power of the resource-restricted WSN. Some research advocates splitting WSNs into lesser nodes and more powerful nodes, around which the other nodes are gathered, cluster heads. The lesser nodes deal with lower level sensing tasks, while cluster heads deal with higher level processing tasks. This generates an asymmetrical network that puts too much pressure on the cluster heads. Even if the cluster heads have much higher processing power and are not energy restricted (they are connected to a power outlet), this deviates from the concept of a WSN as well as not being as applicable. Hence the following related work is picked from those articles that either treat a WSN as homogenous, or as a heterogenous network in which there isn't a lot of difference in processing power between types of nodes.

Research that is covered here generally follows certain steps:

- Make general assumptions about how the network operates.
- Assume certain formulas to calculate chosen metrics.
- Schedule/allocate tasks.
- Simulate the network.
- Plot in relation to chosen performance metric, with the simulation data obtained.

2.1 EcoMapS

EcoMapS(***E**nergy-**co**nstrained Task **Map**pping and **S**cheduling*) is a scheduling system aimed to map and schedule tasks of an application with minimum schedule length subject to consumption constraints[TEÖ07]. EcoMapS is application-independent, as it uses simple tasks with data dependencies noted in a Directed Acyclic Graph (DAG).

The way EcoMapS considers that a WSN operates is based on a few assumptions:

- The WSN for EcoMapS is considered to be formed from multiple clusters of homogeneous wireless networks.
- The algorithm is set to work in one of these clusters.
- Moreover, each of these clusters has a single-hop topology, each node can communicate with any other node in the cluster directly.
- Each cluster executes the application that is setup beforehand, or sent over the air by base stations.
- Communication and processing withing the cluster head is scheduled by the cluster head.
- Time synchronization is believed to be available.
- Sensor nodes can process information and communicate wirelessly at the same time.
- Communications within a cluster do not overlap with communications in other clusters, due to different wireless channel.

2.1.1 Application modelling

Inter-dependence between tasks in the EcoMapS system is described with a DAG, which is a graph with directed edges where each edge e_{ij} means that task v_j depends on task v_i, where $v_i, v_j \epsilon V$, the set of tasks in the Directed Acyclic Graph. The semantics associated with this graph is that a task v_j cannot start execution until all its immediate predecessors (tasks v_i where e_{ij} is a directed edge in the graph) finish execution. Furthermore, the directed edge hints to the existence of data dependency between two tasks. If the two tasks adjacent to the edge are on the same node, execution for the second task can begin immediately, otherwise it must wait for the transfer of results from the first task to complete before starting. This DAG is considered to have only one exit-task (a task with no successors), and as many entry-tasks (tasks with no predecessors) as needed.

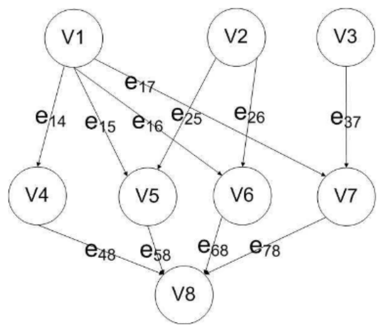


Figure 2.1: *A task **D**irected **A**cyclical **G**raph*

The energy to transmit and to receive information wirelessly is given as a function of l, the number of bits, and d, the distance between the two nodes that are communicated, d being smaller than the largest distance of communication possible.

$$E_{tx}(l,d) = E_{elec} \cdot l + \varepsilon_{amp} \cdot l \cdot d^2 \tag{2.1}$$

$$E_{rx}(l) = E_{elec} \cdot l \tag{2.2}$$

The energy consumption of executing N clock cycles with clock frequency f is considered to be:

$$E_{comp}(V_{dd}, f) = NCV_{dd}^2 + V_{dd}(I_o e^{\frac{V_{dd}}{nV_T}})(\frac{N}{f}) \tag{2.3}$$

The problem to be solved is the mapping/scheduling of tasks, that is to find the set of assignments from task to node and the sequence of execution that properly minimizes the objective function, which can be energy consumption or schedule length. Thus for each task (v_i) we need an assigned node, m_k, with starting time s_{i,m_k} and finish time f_{i,m_k}, execution length t_{i,m_k} and cost (representing energy consumption), c_{i,m_k}. An assignment is a tuple containing these value, $(v_i, m_k, s_{i,m_k}, t_{i,m_k}, f_{i,m_k}, c_{i,m_k})$, noted as h_i. Let $H^x = (h_1^x, h_2^x, ..., h_n^x)$, a schedule, with x denoting the solution space. The result of the scheduling is to obtain $H^o \epsilon H^x$, a schedule which is optimal, meaning it has minimum schedule length under energy consumption restraints.

The problem is then formulated as: min $length(H^o)$, subject to $energy(H^o)$, where

$$length(H^o) = \max_{i,k} f_{i,m_k},$$

and

$$energy(H^o) = \sum_{i,k} c_{i,m_k} \leq EB,$$

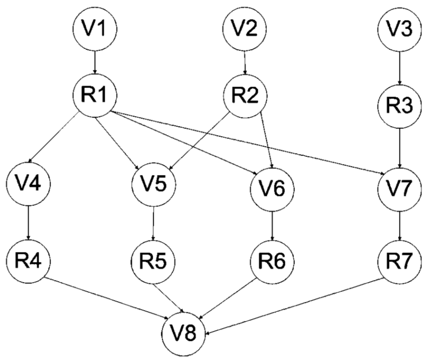


Figure 2.2: *A Hyper-DAG*

EB being an "Energy Budget".

To model the communication channel in a cluster, the authors of EcoMapS use a virtual node C that can execute communication tasks. This is possible because at any given time, only one communication can take place, therefore the network topology can be converted from mesh to a star topology, where each node can communicate only with the center node C. Because communication costs are accounted for in the communication tasks themselves, cost to communicate from a normal node to C is zero. Broadcasting is modelled as communication from node to C, after which every node becomes a potential receiver.

The next step is to include these communication tasks in the DAG. One way to do this is to add more nodes to the graph, nodes that model communication tasks. Therefore, the new DAG, named Hyper-DAG, has a set of nodes $V' = V \cup R$, where V is the set of computing tasks and R is the set of communicating tasks. Each processing task v_i has only one successor, R_i, the communicating task associated with delivering the results to tasks dependent on v_i. R_i has all the successors there were previously attributed to v_i. This new graph incorporates channel access conditions and broadcasting, along with the computing tasks.

The *Dependency Constraint* is therefore reformulated for this type of DAG: *If a computation task v_j scheduled on node m_k depends on a communication task v_i on another node, a copy of v_i needs to be scheduled to m_k, and v_j cannot start to execute until all of its immediate predecessors are received on the same node.*

Scheduling communication in a single-hop network is done by duplicating the communication task from the sender node to C, then to each of the receivers (in the case of a broadcast transmission).

Because of the model adopted, in the process of task scheduling, several constraints have to be satisfied:

- A processing task cannot be assigned to the virtual node C,

$$\forall v_i \epsilon V,\ c_{v_i,C} = \infty,\ t_{v_i,C} = \infty.$$

- A communication task can be assigned both on processing nodes and on the virtual node.
- If a communication task has its immediate predecessor and all of its immediate successors on the same node, it has no energy cost or execution length (since no wireless communication needs to take place).
- Other communication tasks have the energy estimation from formula (2.1).
- If $v_i \, \epsilon \, V$ (computational task) and $pred(v_i) \neq \phi$, then

$$pred(v_i) \subset T(m(v_i))\, and\, s_{v_i,m(v_i)} \geq max f_{pred(v_i),m(v_i)}$$

2.1.2 E-CNPT

There are two phases described in the EcoMapS algorithm: The *Initialization Phase* and the *Quick Recovery Algorithm*, where the former is a static scheduling of tasks involved in an application, whereas the latter is run when one of the nodes (not the cluster head) fails. Constraints in place, the problem of initial scheduling is NP-complete, such that heuristic algorithms are needed to obtain polynomial-time solutions.

The *Initialization Stage* follows somewhat the CNPT algorithm[HJ05], having two stages: *listing stage* and *sensor assignment stage.* The overall strategy is to assign the tasks along the most critical path to the nodes with the earliest execution start time (EEST).

The listing stage consists of sequentializing the tasks into a queue L such that the most critical path comes first and each task is positioned after all its predecessors. First, the Earliest Execution Start Time is calculated for all tasks. The entry-tasks have $EEST = 0$ and $EEST(v_i)$ is calculated by propagating the following formula through the Hyper-DAG downward, from entry-tasks to exit-task:

$$EEST(v_i) = \max_{v_m \epsilon pred(v_i)} \{EEST(v_m) + t_m, LST(v_i)\}$$

The Latest Execution Start Time can then be calculated following the Hyper-DAG in reverse direction, from exit-task to entry-tasks, starting with the LEST = EEST for the exit-task and using the following formula:

$$LEST(v_i) = \min_{v_m \epsilon succ(v_i)} \{LEST(v_m)\} - t_i$$

t_i in this case is the execution time of a task $v_i \, \epsilon \, V$ on a sensor node or a task $v_i \, \epsilon \, R$ on C.

The next step of the listing stage is to push the tasks that represent Critical Nodes (tasks that have their $LEST(v_i) = EEST(v_i)$) on a stack denoted as S. Following this, the top of the stack, $top(S)$ is investigated:

- If $top(S)$ has immediate predecessors that aren't on the stack or in the list, push the immediate predecessor with the minimum LEST on the stack.
- Otherwise, pop the stack and enqueue the popped task.

The listing phase ends when the stack is empty, the result is a list L with the order needed. In the next stage, the *sensor assignment stage*, task are dequeued from L and assigned to the sensors with minimum execution time. The extended algorithm enhances this phase, running it several times for a different number of "active" nodes. If the schedule meets the deadline requirements, then clearly it will consume less energy by using less nodes for the same application. The algorithm for one run is outlined in Listing 2.1, the extension simply iterates over the number of nodes, choosing each time a set with one more "active" node.

Listing 2.1: Single CNPT Algorithm

```
while L is not empty
  Dequeue $v_i$ from L
  if $v_i \epsilon R$ /* communication task */
    Assign $v_i$ to node $m(pred(v_i))$
  else if $pred(v_i) = \phi$ /* entry-tasks */
    Assign $v_i$ to node $m_i^o$ with min $EAT(m_i^o)$
  else /* non-entry computation tasks */
    for all computing sensors $\{m_k\}$
      Calculate $EST(v_i, m_k)$:
      if $pred(v_i) \subseteq T(m_k)$
        $EST(v_i, m_k) \leftarrow max(EAT(m_k), f_{pred(v_i), m_k})$
      else /* communication is needed */
        for $v_n \epsilon pred(v_i) - T(m_k)$
          $CommTaskSchedule(v_n, m(v_n), m_k)$
        $EST(v_i, m_k) \leftarrow max(EAT(m_k), f_{pred(v_i), m_k})$
  Keep the schedule with $min(EST(v_i, m^o))$
  Schedule $v_i$ on $m^o : s_{v_i, m^o} \leftarrow EST(v_i, m^o)$
```

- $T(m_k)$ denotes the tasks assigned on node m_k.
- $m(v_i)$ denotes the node on which v_i is assigned.
- $pred(v_i)$ and $succ(v_i)$ are the immediate predecessors/successors of task v_i.
- $EAT(m_k)$ is the Earliest Available Time on node m_k.

The idea is simple: we assign communication tasks to the node where the data-generating task (for that communication, that is to say the immediate predecessor in the Hyper-DAG) was assigned. If the task is an entry-task, we assign it to the node with the earliest available time. If it is a non-entry computing task, we check if we're generating the earliest execution time for the same node, and if we are, mark it as the ending of the predecessor task (in the Hyper-DAG), or the earliest available time on that node (whichever is later). If it isn't the same node, then we need communication: we propagate the communication task to the virtual node and to the recipients. Once the propagation is done, we can once more see when the task can be schedulet earliest. We then schedule the task on the node that appeared to have the earliest time of all.

2.1.3 E-MinMin

A second algorithm developed for the Initialization Phase of EcoMapS is also based on a simpler algorithm, MinMin, and searches for the optimal number of computing sensors that has the smallest schedule length with an energy constraint. For each assignment (v, m), MinMin algorithm calculates a *fitness* function $fit(v, m, \alpha)$. The fitness function expresses the cost in time and energy of assigning task v to node m. α is a trade-off parameter that weighs in for time or for energy.

Listing 2.2: Single MinMin Algorithm

```
for $\alpha = 0$; $\alpha \leq 1.0$; $\alpha += \delta\alpha$
  for entry-tasks $v_i$
    Assign $v_i$ on node $m_i^o$ with $min\, EAT(m_i^o)$
    Assign $succ(v_i)$ on $m_i^o$
  Initialize the mappable task list $L$
  while $L$ is not empty
    for task $v_i \epsilon V$
      for all computing nodes $m_k$
        if $pred(v_i) \nsubseteq T(m_k)$
          for $v_n \epsilon pred(v_i) - T(m_k)$
            $CommTaskSchedule(v_n, m(v_n), m_k)$
        /* propagate communication task */
        Calculate $fit(v_i, m_k, \alpha)$
      Find $m_i^o : fit(v_i, m_i^o, \alpha) = min$
    Find $(v, m) : fit(v, m, \alpha) = min$
    Assign $v$ to $m$, remove $v$ from $L$
    Assign $succ(v)$ on $m$
    Update $L$ with unassigned mappable tasks
Among schedules with different $\alpha$
```

```
  if $\exists H : energy(H) \leq EB$
    Return $H^o : length(H) = min$
  else
    Return $H : energy(H) = min$
```

The main idea behind Listing 2.2 is to map computational tasks, leaving communication tasks on the same node, leaving them to propagate with the function $CommTaskSchedule(,,)$. Among all possible assignments, the one that is most fit is chosen at each step.

2.1.4 Quick Recovery Algorithm

As previously mentioned, WSN have special requirements for scheduling algorithms. Node failure is common in WSNs, so one such requirement is for the algorithm to have failure handling. As rescheduling from scratch is not efficient, EcoMapS proposes a quick recovery solution: assign the tasks of the failed sensor to the sensor with the most idle time.

2.1.5 Outcome

EcoMapS aims to minimize schedule lengths of applications under energy consumption constraints. Using the channel model and communication scheduling algorithm, EcoMapS simultaneously schedules communication and computation tasks. Simulation show that EcoMapS is quite performant in respect to other similar algorithms. E-MinMin based EcoMapS outperforms the other version, although E-CNPT-based EcoMapS has less computing overhead. The former would then be best suited for stable networks, while the other would be for WSNs where updates are more frequent.

2.2 Real-time support in Wireless Sensor Networks

A different aspect of the scheduling problem is scheduling a given set of tasks (can be repeteable) on a single node, taking into account energy efficiency, as done in [DPdR+05]. For the scheduler to be able to make correct, power-aware decisions, tasks that are set to execute on the node must specify a worst execution time and a deadline, as well as an indicator of "importance". This "importance", also denoted as a power index, shows the relative importance of a task in relation to the other tasks under low-power conditions.

The main idea is that to extend network lifetime, non-critical tasks will be scheduled at greater intervals. All tasks have an initial deadline; if the sensor node is in a low-energy state (30% battery left), then the tasks runs a piece of code to redetermine its deadline in respect to the remaining battery life. Extending the deadlines for most of the tasks executing on the node can mean that there's time

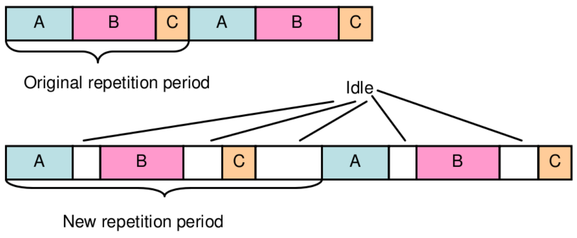


Figure 2.3: *Idle time insertion through deadline extension*

left in which the MCU is idle, which is supposed to consume less power than in the active state, hence the average power consumption is lowered and network lifetime increases.

2.2.1 PA-EDF

Power-Aware Earliest Deadline First is a scheduling algorithm based on EDF that can make power-aware decisions when dequeueing tasks. If the battery level is above a certain threshold, tasks are scheduled in the same way as in EDF. Once the battery level drops below a threshold, tasks with the power-index smaller than the threshold are considered "important" and are scheduled in the same way, while tasks with larger power-index are given a chance to reconsider their power-index, reducing the frequency with which they execute at the same time.

2.2.2 APA-EDF

Adaptive Power-Aware Earliest Deadline First is similar to PA-EDF, the difference being that there is more than one threshold, there are n thresholds, where n is also the number of power indexes possible. Each power index is associated with a threshold. Each time the scheduler dequeues a task, it checks its power index against the current battery remaining and acts accordingly, either it schedules it or executes the special reanalysis code.

2.2.3 Findings

PA-EDF and APA-EDF successfully incorporate the power consumption issue into realtime scheduling, increasing the network lifetime overall by increasing the lifetime on each of the nodes. These algorithms allow the network to focus on the more important tasks, reducing the frequency of less important tasks when batteries are low.

2.3 The SENSEI project

SENSEI (Integrating the Physical with the Digital World of the Network of the Future) is a project integrated in EU's 7th Framework Programme for Research an Technological Development.

SENSEI is meant to help build the network of the future, integrating Wireless Sensor and Actuator Networks (WS&ANs) into a framework of global scale. This framework is envisioned as service oriented, because WSANs are by nature heterogeneous. WSAN islands should be able to offer services through a unified service interface to be integrated more easily in different applications. Network and information management services will ensure correct functioning of these islands and proper adjustment to the ever-changing physical environment. Accounting, security, privacy and trust are also key issues in integrating these islands in the network of the future.

Goals of the SENSEI project are:

- **Scalability of the framework** - The protocols that are part of the framework must support easy integration of very many WS&AN islands into a global network. New networks must have plug and play integration into the greater network, allowing for greater ease of use of WS&ANs.

- **An open service interface** - Unification of application access to information and actuation services. This will help the development of new applications onto already connected/existent WS&AN networks

- **Efficient WS&AN island solutions** - Aim of this being wireless transmission with 5nJ/bit, by using energy-aware protocol stacks, and ultra low-power transceivers. This would greatly increase network lifetime of a WS&AN network.

- **Pan European test platform** - A large scale experimentation and field trials of proposed technologies, evaluating the progress of integration of WS&ANs into the Future Internet.

2.4 Applications of Wireless Sensor Networks

2.4.1 Industrial Control and Monitoring

Wireless sensor networks can be used in industrial environment with a great deal of success. Usually an industrial facility consists of a small control room and the rest of the physical plant. This control room has indicators and displays from different sensors throughout the plant, as well as inputs for actuators. Instead of using expensive thick cables to connect the control room to the sensors and actuators, a wireless sensor network can be used.

The conditions for this type of network would be that reliability has to be high, conditions that can be satisified by a network with many nodes and many possible routing paths from each node to the data sink (positioned or connected to equipment in the control room).

Possible applications here are in industrial safety, for instance, a network that detects presence of noxious, poisonous or dangerous materials on a plant. Significant notice needs to be given to WSNs with self-healing and self-maintenance, which will be resilient enough to be able to gather data in the event of an explosion/damage on the plant.

Another use of WSNs would be in monitoring or control of rotating/moving heavy machinery, reducing the possibility of accidents due to collisions. A wired solution would be undesirable for this application, for instance in the case of a rotating piece of equipment wires would hinder its freedom of movement.

Present HVAC systems (Heating, Ventilation and Air Conditioning) suffer from lack of data, they generally have a few thermostats per building. The end result is a large difference in air temperature in the building between different rooms. A wireless sensor network would be able to cover a lot more space, gathering enough information for the system to be truly efficient. A smart HVAC system, with the help of an appropriate sensor system, would know that activities in one room keep it hot, while activities in another room (or positioning in relation to cardinal directions) keep it cool, and would be able to correctly decide how to circulate air between them, keeping them cool in summer and warm in winter.

2.4.2 Home Automation

Wireless sensor networks are particularly useful in homes, where adding wires is not always easy. A WS&AN inside the home could communicate with a "universal" remote control (for electronic devices, curtains, locks, etc.). Another concept that is related to home automation is Remote Keyless Entry (RKE), either for the home itself or for the car.

Monitoring room temperatures can be applied in home automation as well, wireless temperature monitoring can be a good replacement for a normal thermostat, and would not need any extra wiring. The heating system could then maintain the same temperature throughout the house, room by room. It could also discover room where insulation needs to be upgraded. Rather than upgrade insulation for the entire home, which is costly, the owner could then upgrade only the insulation responsible for the heat loss. Wireless sensor networks make intelligent home monitoring systems more accesible and more powerful.

2.4.3 Security/Military Sensing

WSNs can act as sentries around defensive perimeters, or can be used to identify and locate targets on a battlefield. They are easy to deploy on the battlefield and can be well hidden (camouphlaged as rock, mounds or what-not), are robust because there is no single point of failure (with multiple alternate routing paths), making them difficult to destroy in battle.

Target tracking is a good example of a collaborative application of WSNs. Sensor nodes must take measurements then process them distributively to find the location of moving targets or classify targets.

2.4.4 Asset Tracking and Supply Chain Management

An example of asset tracking with WSNs would be an intelligent warehouse, where each object can be easily found because it has a node with an id. This boosts the efficiency of the warehouse, because a lost object can be a lost sale, while taking the same costs by occupying space in the warehouse. Management of large objects can indeed be aided by the use of a WSN, as with railroad cars in railyards, or shipping containers in a port. Managing containers is a lot more sensitive since the containers are stacked onto one another, and removing a container at the bottom is a costly operation. Having precise information about the location of each container needed makes way for the use of an algorithm such that the entire process can be optimized.

2.4.5 Intelligent Agriculture and Environmental Sensing

A common known fact is that water is a major factor for plant growth. While too little water means reduced yields, too much leads to drainage and diseases. Thus a very important part of agriculture is water-management. What WSNs give is the ability to make localised decisions instead of global ones. Field conditions are not uniform, and treating the problem as if they were means that some portions of the field might be under-irrigated, while others might suffer from excessive application. Wireless rain gauges and humidity sensors could monitor the status of the field, as well as control individual sprinklers in a zone. This leads to a system that is water-efficient, energy-efficient, low-cost and autonomous, as demonstrated in [YJK07]. Use of WSNs can be extended to use of chemical and biological sensors to asses the need for pesticides, herbicides and fertilizers, or to search for contaminants such as mercury.

As for Environmental sensing, the main use is for early detection of natural disasters such as floods, earthquakes or forest fires, as well as monitoring grand scale fenomena, i.e. glacier movement. A WSN to detect forest fire could be deployed by releasing the nodes from a helicopter, thus reaching in remote places. Nodes would have to be self-sufficient or have enough network lifetime to be worth using. WSN nodes with temperature and humidity sensors could be clustered in the forest around a special gateway node that could communicate the data with a mobile network access technology (e.g. UMTS).

WSNs can help minimize damage produced by an earthquake by assessing the structural integrity of a building. Nodes would be scattered along the walls and in critical spots and they would measure structural responses to external vibrations, inferring the structural integrity.

2.4.6 Health monitoring

WSNs is starting to invade the non-life-critical health monitoring. This means athletic performance monitoring, an athlete carrying a BSN (Body Sensor Network) will execute his exercise then he will be able to assess both his performance and his body's response to the effort. Professional and semi-professional athletes have already been using these wireless network to monitor pace, heart rate and distance, but the technology should start covering recreational athletes as well.

Inside the hospital WSNs could form a night watch assistance system, were non-life-critical parameters such as temperature could be monitored on a set of pacients, reducing the need for the nurse to check their health status constantly.

Real medical applications will be slow in emerging, due to regulations and very high need for robustness and accuracy.

Chapter 3

Hardware Platform

In this chapter we will present the hardware platforms used in the development of the algorithm. The variety of platforms was a key point in designing the scenario to test the algorithm, as this demonstrates that task modelling of application abstractizes beyond platforms, making heterogenous wireless sensor networks such as these possible.

We will discuss the platforms in chronological order of use, our WSN started off with the Raven development kits. The next step was for our team to develop our own wireless sensor nodes based upon wireless modules (the Zigbit module), and these are the Sparrow nodes. The Power Sparrow nodes inherit much of the original, but have very specific purpose, measuring power usage of appliances.

All platforms run the Contiki Operating System and will have the same core of software functionalities. Additional capabilities will be given by the presence of sensors attached to the platforms. Other differences will be in measuring the power used by system, measured by the system itself. There will be inherent differences as to how this is handled on each platform, mainly because of the hardware design and technology used, as well as expansion connectors provided.

The heterogeneity of the platforms must not be seen as similar ways to do the same thing, each platform best suits a specific purpose. The Raven nodes are an all-in-one sensor boards, with user interface, Sparrow nodes have an expansion board for energy harvesting experiments, Power Sparrow measure outlet voltage, powering themselves directly from the outlet.

3.1 The AVR Raven development kit

The AVR Raven developed by Atmel is a development kit for the AT86RF230 radio transceiver. A kit usually consists of two AVR Raven wireless nodes and one Raven USB stick that acts as a gateway. The stick is connected to a PC (or embedded system) and acts as a remote wireless card, offering an additional interface to communicate with the nodes.

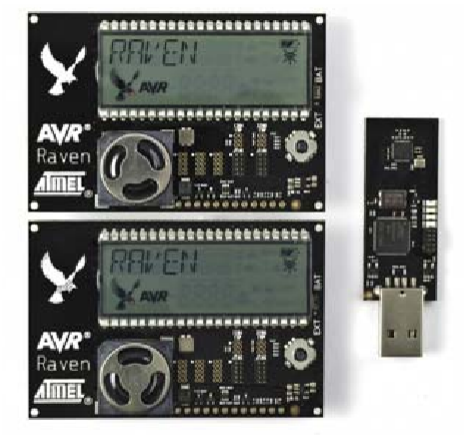


Figure 3.1: *The AVR Raven development kit*

This forms a versatile kit that can be used for debugging or building a variety of RF applications in the 802.15.4 frequency range: from point-to-point communication to large WSNs(using more than one kit). Using more than one Raven USB Stick, even multi-sink networks can be formed, apart from the usual single-sink networks that can be built with only one kit.

3.1.1 The Raven Nodes

Individual Raven Nodes are built around two microcontrollers, the Atmega1284P and the Atmega3290P, the former dealing with radio communication and the latter with the user interface peripherals. These microcontrollers are selected from the picoPower family, ensuring minimal power consumption and viable operation down to low supply voltages (1.8V). The Raven Node communicates wirelessly via a transceiver chip (the AT86RF230) and has a built-in PCB antenna.

Communication between the two microprocessors and between the Atmega128 and the radio transceiver is accomplished through a synchronous serial interface (**USART** - Universal Synchronous Asynchronous Receiver Transmitter).

The Atmega1284P holds the wireless stack code and acts upon user input on the other microcontroller. It uses a 32.768 kHz oscillator for a realtime clock, but its main clock is internal, set at 4MHz.

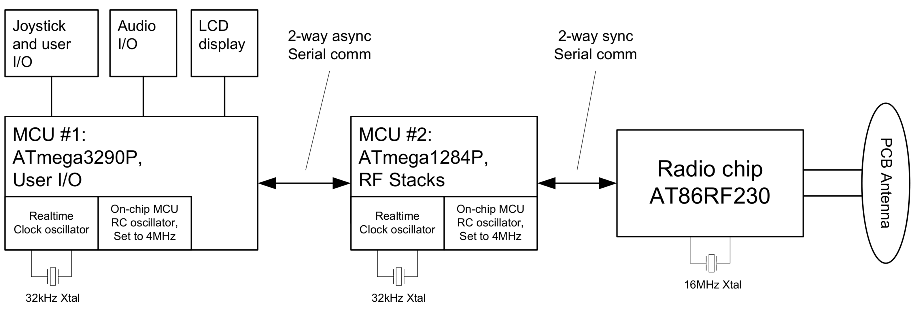

Joystick and user I/O
Audio I/O
LCD display
2-way async Serial comm
2-way sync Serial comm
MCU #1: ATmega3290P, User I/O
Realtime Clock oscillator
On-chip MCU RC oscillator, Set to 4MHz
MCU #2: ATmega1284P, RF Stacks
Realtime Clock oscillator
On-chip MCU RC oscillator, Set to 4MHz
Radio chip AT86RF230
PCB Antenna
32kHz Xtal
32kHz Xtal
16MHz Xtal

Figure 3.2: *Overview of an AVR Raven node*

The Atmega3290P drives the Raven LCD, has audio output/input and a joystick as human interface. It also maintains a 32.768 kHz oscillator for realtime clock purposes besides its main clock of 4MHz. Audio output on the node is controlled with PWM, whose signal passes through a low-pass filter and a class-D amplifier before reaching the 8Ω speaker. A microphone is connected to the node for audio output, the signal is amplified and low-pass filtered before reaching one of the ADC (Analog-to-Digital Converter) pins on the chip.

Each microcontroller has a chip with persistent memory attached, the Atmega1284P has a 2Kbits Serial EEPROM (AT24C02B) attached on TWI (Two-Wire Interface). This memory holds configuration and calibration data that are write-protected. The Atmega3290P has a 16MBits Serial Dataflash attached. This memory is designed to hold firmware images as well as sounds. It will operate however only when the supply voltage is above 2.5V, unlike the rest of the system (If batteries are low the Raven node might still be able to transmit but it will not be able to read/write data from the external dataflash.

The PCB antenna is a folded dipole antenna of 100Ω with a net peak gain of 5dB.

The power supply for the node can be internal (LR44 batteries), or external, with supply voltages ranging from 5V to 12V. The external supply voltage also passes through a voltage divider and is connected to 2nd ADC pin of the Atmega3290P microcontroller, therefore the node can monitor the external operating voltage.

The Raven node exposes both JTAG and ISP programming connectors, together with several GPIO headers.

3.1.2 The Raven USB Stick

The Raven USB Stick is built around a USB-capable microcontroller and the same AT86RF230 radio transceiver chip. The USB-capable microcontroller is the AT90USB1287, capable of acting as a full speed USB device. With the

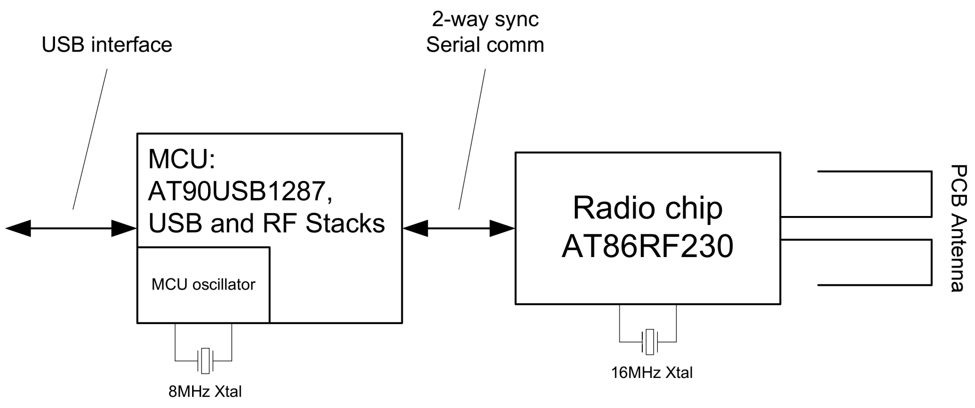


Figure 3.3: *Overview of an AVR Raven USB Stick*

Contiki Operating System, the USB Stick acts as a gateway for the WSN and as a remote network interface (RNDIS) on the computer it is connected to.

The antenna on the Raven USB stick is a dipole antenna with a net peak gain of 0 dB.

It has a JTAG programming interface and a serial interface for "printf" debugging, along with 4 LEDs to show current status.

3.2 Sparrow Wireless Sensor

The sparrow is a sensor node built around the powerful Zigbit™module. It provides a JTAG header for programming, and expansion connectors for two USARTs (Universal Synchronous/ Asynchronous Receiver Transmitter), one TWI (Two-Wire Interface) and 3 ADC channels (Analog-to-Digital Converter).

For user interaction there is a switch and a couple of leds. As sensing capabilities, it has a SHT11 humidity and temperature sensor connected by a software SPI (Serial Peripheral Interface).

3.2.1 Zigbit™module

The Zigbit™module is an all-in-one wireless module, consisting of an Atmega128 microcontroller, a RF transceiver chip and a dual chip antenna, being perfect for small-footprint wireless sensor applications. It is designed to operate in the 2.4GHz band, as a 802.15.4/Zigbee node. Its high sensitivity and low-power guarantee fitness for any wireless application.

The mixed-signal small-footprint package make it ideal for rapid prototyping and research endeavours.

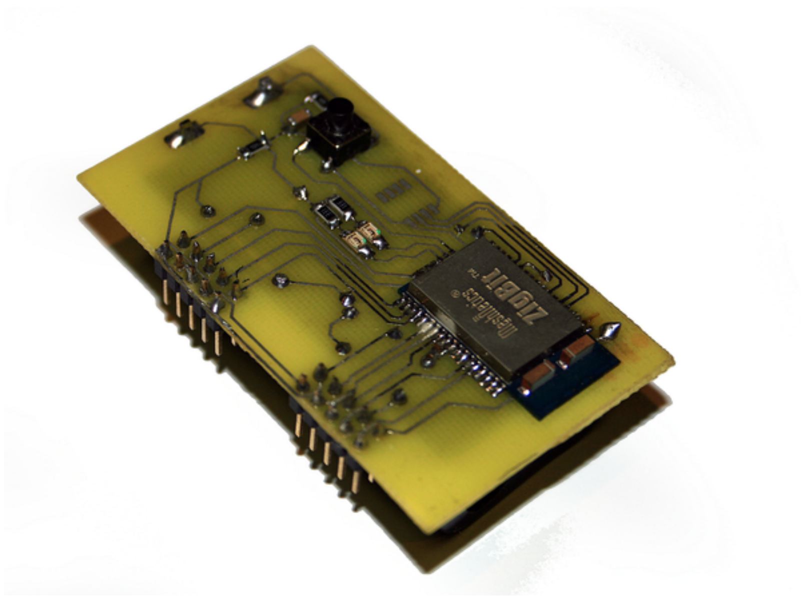


Figure 3.4: *The Sparrow wireless node*

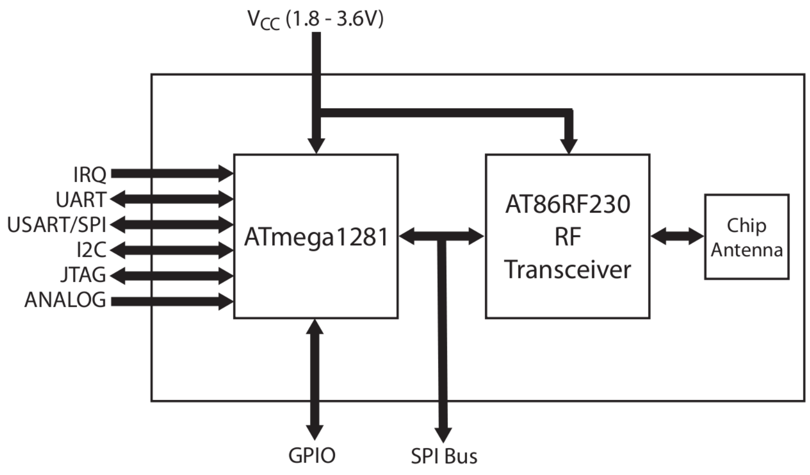


Figure 3.5: *Overview of the Zigbit™module architecture*

3.3 Sparrow Power

The Power version of the Sparrow nodes are built so that they can obtain their power directly from the power outlet they will measure. The wireless part still uses the Zigbit™module,but additional circuitry has been added to be able to sample voltage and current at the outlet.

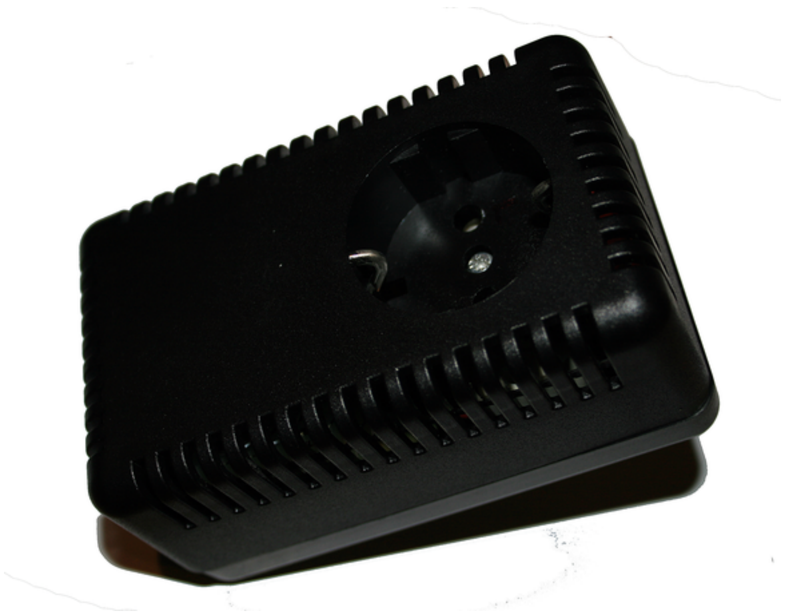


Figure 3.6: *The Sparrow Power wireless device*

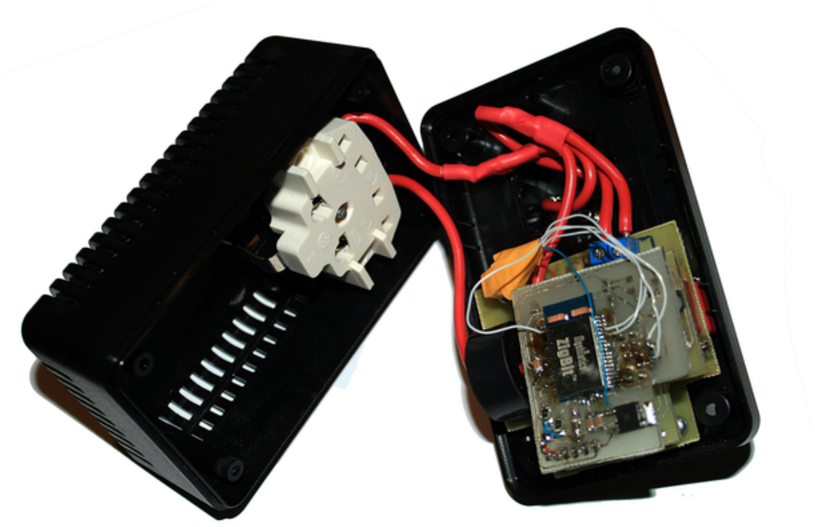


Figure 3.7: *The Sparrow Power wireless device(inside)*

Chapter 4

Software Environment: Contiki Operating System

The Contiki Operating System is an open-source, highly portable multi-tasking operating system for memory-efficient networked embedded systems and wireless sensor networks. It is our choice at the moment as platform for this study. It offers IP communication, both IPv4 and IPv6. Currently the IPv6 implementation is in IPv6 Ready Phase 1, which means that some functionality related to neighbour discovery is not complete, leading to a lack of multi-hop routing. Platforms supported by this operating system range from MSP430 from Texas Instruments and AVR Atmegas to old home computers.

The innovations introduced by this system were the introduction of IP communication in low-power wireless sensor networks, with the uIP stack and the protothreads system, introduced in 2005. The best feature is the implementation of a network interface on the Raven USB Stick, such that nodes can be communicated with transparently from outside the wireless network.

Contiki offers a collaborative multi-tasking environment, based on a coroutine mechanism, in which each process must at one point yield the processor so that a new process is scheduled. Contiki has some hardware abstraction, especially for commonly used part. Network communication is abstractized with a “protosocket”, and libraries exist for energy estimation, timers, event timers, etc.

4.1 Process kernel

4.1.1 Protothreads

Protothreads are an innovation made for the Contiki system, although the system is based on the coroutines implementation in C, which have been around for quite some time. In the Contiki operating system, they are called *local continuations*, allowing a subroutine (a function) to pause and resume execution of its code. These local continuations are formed with normal function that call some defined macros.

Listing 4.1: Coroutines in C (Simon Tatham)

```c
int function(void) {
    static int i, state = 0;
    switch (state) {
        case 0: /* start of function */
        for (i = 0; i < 10; i++) {
            /* so we will come back to "case 1" */
            state = 1;
            return i;
            /* resume control straight after the return */
            case 1:;
        }
    }
}
```

The code in 4.1 demonstrates the use of coroutines, the repeated call of function() will yield results from 0 to 9. First it will change internal state from 0 to 1, posting a new label for the switch statement to jump to. Based on this example, one can easily define suggestive macros to generalize:

Listing 4.2: Coroutine macros in C (Simon Tatham)

```c
#define crBegin static int state=0; switch(state)\
                                { case 0:
#define crReturn(i,x) do { state=i; return x;\
                               case i:; } while (0)
#define crFinish }
```

Indeed, the local continuations in Contiki are similar:

Listing 4.3: Local Continuations

```c
typedef unsigned short lc_t;

```

```
#define LC_INIT(s) s = 0;
#define LC_RESUME(s) switch(s) { case 0:
#define LC_SET(s) s = __LINE__; case __LINE__:
#define LC_END(s) }
```

The protosocket library build on top of this, adding yield functionality and the possibility to "block" waiting for a condition. To do this each protothread holds an `lc_t` structure together with a semantic of how it was interrupted (`PT_WAITING`, `PT_YIELDED`, `PT_EXITED`, `PT_ENDED`). Yield means giving up the processor, waiting is yielding until the protothread is scheduled and a condition is met, exited resets the state of the protothread (of the local continuation on which it is based), end means the protothread has reached the end of the function and the state is reset (not too different from exit).

4.1.2 Processes

Processes are built upon the protothread structure, each process having one protothread. The functionality added to protothread is of context and higher-level abstraction. Processes in Contiki run as a result of responses to *events*. An event can be triggered by a timer, by the uIP stack (network event) or even by the process itself (a continue event used by a pause function in the process library).

Listing 4.4: Process definition

```
#define PROCESS(name, strname)                                    \
  PROCESS_THREAD(name, ev, data);                                 \
  struct process name = { NULL, strname,                          \
  process_thread_##name }
```

The scheduler is a simple queue on which events are posted. When an event is popped, the associated process is called with the specific event. The process is therefore a complex piece of code that retains state (through local continuations) and reacts on new input (through event data). This adds versatility to the already existent protothread library.

Commonly, a process will have after declaration a specific thread function, its definition must begin with the macro `PROCESS_THREAD`, it needs `PROCESS_BEGIN` and `PROCESS_END` macros and can then use wait, yield, pause macros for the control flow of the operating system (the responsiveness of the entire system depends on each process running, latency can very quickly become a problem because of a slow process).

4.2 Tasks in the Contiki context

There are two possibilities regarding the implementation of generic tasks, one is to have them all run under a single process, using timer events and with a scheduler similar to real-time operating systems. A timer would be set to expire when the first task would be due. The disadvantage to this method is a difficulty in coming up with a metric for processor use (versus idling).

The other possibility is to treat a task as a process. There would be a task manager that would be able to start and stop tasks, alter settings, append subscribers, etc. A performance metric can be calculated using this method concerning the scheduler. One of the tasks would measure the timeframe between two succesive schedules of the same task. As the system keeps more and more tasks running (as opposed to waiting, when they are still scheduled), and more and more tasks use more CPU time, time between schedules will increase.

Starting the task would be similar to starting a process in Contiki, while stopping a task means marking it as a WAITING process. While it is marked thus, in its (presumed) main infinite loop it would wait until it is taken out of this state.

Obtaining information from the task can be done in two ways:

- Collecting data directly, with get/set ¡parameter¿
- Data sink method, where the entity that connects to the sensor can register itself or another entity as a data sink for the output that is generated from the task. Parameters can be given such as the frequency with which data is forwarded to the subscriber (division of the frequency with which data is generated

Listing 4.5: Task definition

```
#define TASK(title , process)\
PROCESS(process, title);\
static struct task_list_t el_##process ={\
        .value={\
                .name=title,\
                .status=WAITING,\
                .app=&process}\
}
```

A list of processes can be kept with the linked list API available in Contiki. The structure that represents the task would be:

Listing 4.6: Task structure definition

```
typedef struct
```

```
{
        struct process * app;
        uint8_t status;
        char name[10];
        subscriber_list_t subscribers;
} task_t;
```

4.3 Energest - Energy Estimation Module

Energest is at the base of the *Communications Power Accounting* power in the Contiki Operating system. The aim is to present proportional data regarding the types of wireless transmission taking place, with the scope of finally determining which component uses more power. Energest supports multiple levels of accounting for energy tasks, and has a counter for each. Time is measured with a real time clock between checkpoints and it is added to the counter of the active levels. This is how the communications power accounting measures transmit and listen times, and will be useful for MAC protocols that have power management. However, the stack on the AVR does not support this, so we will opt for a hardware alternative of measuring power, as shown in 5.5.

Chapter 5

Static scheduling

A first step in task scheduling would be to consider the static case, where all the tasks that have to run are known from the beginning. An assignment to nodes would take place, then the results will be distributed to the nodes. The system will not allow at first the adding or removal of tasks. This static case will have a more efficient scheduling because of its conditions, so it is to be preferred to the dynamic one if we already have all the prerequisites. Indeed most applications will be a mixture of static and dynamic functionalities, so the tasks belonging to the former will be scheduled in the beginning, more efficiently, then the one from the latter will be scheduled as need arises.

5.1 Assumptions

We have discussed the hardware and software platform, and now we must note some characteristics of how they relate to the scheduling problem. First of all the network in this phase can only have star topology, as this is all that supports Contiki's UIP stack. Only the Raven USB stick with Contiki can act as a router. This is not very relevant as behaviour in a larger, multi-hop network relies on this basic star topology. One important feature is that nodes can send messages to one another directly.

When we will measure the energy consumed running a task we will consider that energy consumed in idle time is negligeble. However, Contiki does not have any power management on the stack level, so power is wasted even in idle state. This should not affect the results on the scheduling algorithm, as the energy relevant to the problem is taken into account.

Some nodes have extra peripherals attached, such as LCDs or LEDs or speakers. This will appear as extra consumption on these nodes, so the scheduler will not favor them as much when choosing an appropriate node for a task. This is not a problem, on the contrary, it should demonstrate the usefulness of the scheduler in a heterogenous wireless network.

5.2 Energy considerations

Energy efficiency is a key issue in WSN applications and is no stranger to this study. WSNs have limited, irreplaceable energy sources (most of the time, wireless nodes that incorporate any form of energy harvesting take exception from this fact) but have harsh lifetime requirements. In order to formulate criteria for scheduling to minimize energy consumption, we have to discuss how energy is consumed in a WSN.

The main culprit in energy consumption is radio communication. Energy spent on time communicating on wireless is about a couple of hundreds up to a thousand times larger than the energy spent processing on the microcontroller. Thus energy savings efforts must go into keeping the wireless chip in low-power states as much as possible.

Another interesting fact is transmission versus reception power. Both in the transmit state and the receive state, the RF part of the transceiver is active, so it may seem that the two are energy-equivalent. However, due to differences in hardware design, the transceiver will consume more power in the receive state. Although transmission times are kept low because transmission generally occurs in small bursts, the nodes must be able to receive messages at all times. Even if a proper communicating schedule is put in place, receive timer will still be much larger.

Channel sensing, to be used in CSMA (Carrier Sensing Multiple Access), has about the same energy requirements as receiving and transmitting, making MAC protocols that abuse it energy inefficient.

Energy savings could be done with power control over the transceiver, reducing the detection and communication radii. However, this only reduces power consumption at almost half the initial value, which is not considered to be a relevant energy saving in respect to the need to put a lot more nodes to compensate for the small radius of communication.

5.3 Problem Definition

The problem we address is an unconventional scheduling algorithm, in the sense that the main constraint is not time, but energy. As shown in Related Work, research on scheduling is generally focused on hard time deadlines. Instead, we propose a solution where time is of least importance, preceded by energy consumption, battery awareness, availability and affinity.

The task that we wish to schedule is the smallest indivisible part of an application. Tasks can be classified into sensing tasks, actuating tasks, detection tasks, etc. For example, we have a fire detection system implemented with a WSN. We can have smoke sensing tasks on nodes that have smoke sensors, an event detection task, that detects in a stream of sensor input when smoke levels have risen, and an alarm task, that handles the behaviour of the network in the case of fire (bell, speaker, opening doors, etc.).

For the previous scenario, the scheduler needs some basic information for each task: its importance, an affinity to a certain type of node (a smoke sensing task can only be assigned to wireless nodes that have a smoke sensor), a frequency with which to run (most if not all tasks in a WSN are repeteable) and data sinks (addresses and ports). In the fire detection scenario, one setup can be the following: some nodes can have smoke sensing tasks with a certain frequency (e.g. once every minute),[*] situations the tasks have a single data sink (different from the gateway) on a node that does not have a smoking sensor. The event detection task is assigned on that node, it runs an algorithm on the data received from all the nodes and determines if there's a fire and where it would be. The output of this task is handed over to nodes that handle the alarm (multiple data sinks for this task). Each node that has an alarm or actuates a device in case of fire is registered with that event detection task. When it receives positive confirmation of a fire, it triggers the alarm.

Parameter	Before scheduling	After scheduling
Importance	✔	
Data sinks		✔
Frequency	✔	✔
Node Affinity	✔	
Multiplicity	✔	✔
Data dependency	✔	

Table 5.1: Origin of task properties

This setup is scalable, meaning there can be more than one node running the event detection task, assuring minimal energy consumption. So another parameter for a task that has to be scheduled is its multiplicity, which can be a number or left to the scheduler to adjust[†]. Data dependencies between these types of tasks (smoke sensing, event detection and alarm) should be clear from the start.

[*]A variation of the algorithm that takes into account low-battery states for nodes or just where a task can't be scheduled due to network overencumbrance can be implemented, similar to[DPdR+05]

[†] Another possible variation of the algorithm is when the number of task to run in the network is determined by the scheduler itself. The ultimate goal of the application needs to be taken into account, and some tasks are better left to run on more nodes for the sake of the application or of energy awareness

To summarize, the properties of a task and the moments when they can be determined are shown in Table 5.1. Notice that some can be determined in both moments, according to the variation of the algorithm.

Thus the problem is to schedule a number of tasks, having the WSN and its topology as input, and the tasks' frequencies and data dependencies. The scheduler finds the best assignment that maximizes network lifetime and achieves the application goal. Results are evaluated by calculating the total energy cost of the tasks running in the WSN.

5.4 Initial Metrics

5.4.1 Battery Level

The first important metric that needs to be taken into account when scheduling tasks is battery status. We will use the Atmega Analog to Digital Converter (ADC) for this purpose, however this cannot be accomplished by measuring the ADC0 pin on the Atmega3290 on the Raven board, because there is a reference voltage of either 1.1V (internal) or VCC, so it will always read maximum. The solution is to connect the power supply to one of the voltage measurement inputs, one that already has a voltage divider, which is connected to the ADC3 pin.

$$ADC_3 = \frac{100K}{471K} \cdot VCC \ < VCC$$

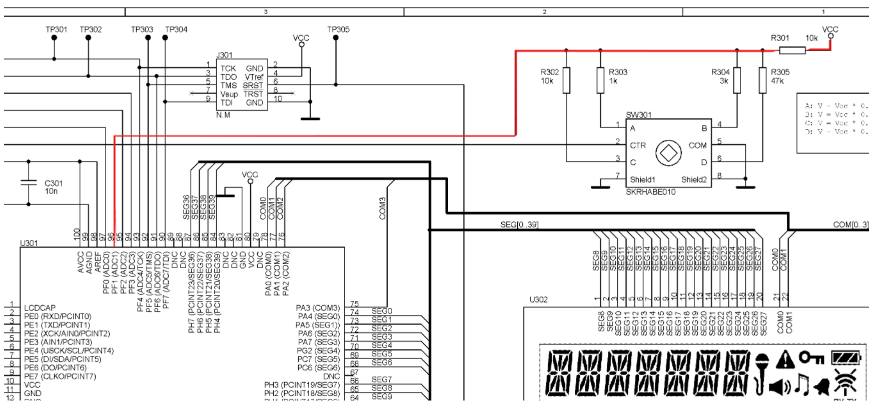


Figure 5.1: *ADC0 voltage measurement highlighted on the AVR Raven schematic*

The ADC is available only on the Atmega3290 microcontroller so the information has to be transferred to the Atmega1284. This is done once every 10 seconds, when the Atmega3290 sends a frame (Contiki has an additional communication layer on top of the existing inter-processor USART) with a SET_BATT command, and with a unsigned 16-bit integer as payloadi (broken into bytes). The integer sent is exactly the ADC conversion result, right-aligned (only the 10 least significant bits are used). The integer is recomposed at the other end and saved in the data memory. We will discuss later how this information reaches the scheduler service. The battery indicator on the display LCD of the Raven board is also used to display status.

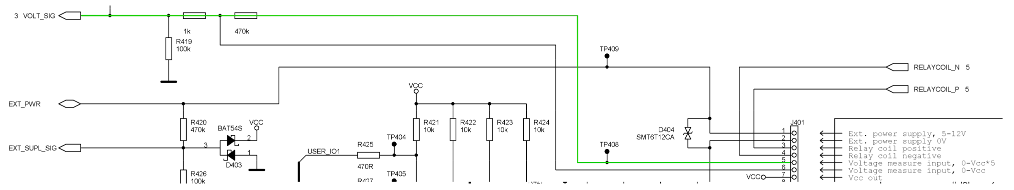


Figure 5.2: *ADC3 voltage measurement through voltage divider on PCB highlighted on the AVR Raven schematic*

5.4.2 Node affinity

In this section we define the *Node affinity* for our scheduler: Each task can only run on certain nodes, as was previously discussed. For this reason we create the affinity parameter, which is 0 for assignments (v, m) (task v and node m) that are imcompatible and 1 for compatible assignments. We might also define this notion from the nodes' point of view. which further illustrates the connection between Node Affinity and its implementation: Each sensor node has a specific set of capabilities: It can run sensing tasks based on attached sensors, or actuating tasks that are compatible with actuators attached, diverse computing tasks and so on.

This is implemented on the platform by compiling into the board code some Contiki processes, while others are left out. This means that each node is aware of the processes it contains, and can advertise that to the scheduler. What happens is that each node will have several *metaprocesses*, processes that help the scheduling, know which processes exist on a node and can start them upon command. One of these processes will advertise to the scheduler (to the network node on which the scheduler runs, in this case a PC) the node parameters, its battery status and capabilities.

Let C_m be the set of capabilities advertised by a sensor node. Each task has a set of capabilities needed in order to run (a smoke sensor for a smoke sensing task), let this be C_v. Node affinity in the case of a possible assignment (v, m) is 1 if $C_v \subset C_m$, and 0 otherwise. Node affinity for a task (not an assignment) will be the set of nodes M, such that $\forall m_i \epsilon M$, $NA((v, m)) = 1$, where $NA((v, m))$ is Node Affinity for the assignment of task v to node m.

5.4.3 Node load

As described in 4.2 we can measure node load by looking at the time between schedules of the same dummy-task. This is especially useful when dealing with processing tasks outside the task context. However, the tasks are periodic so they only run when their event timer expires. Then the metric would be reduced to showing the number of processes actively running and processing. This has to be combined with a normal calculation of tasks and their frequencies to tell if a node can run another task or not.

5.5 Assessment Metrics

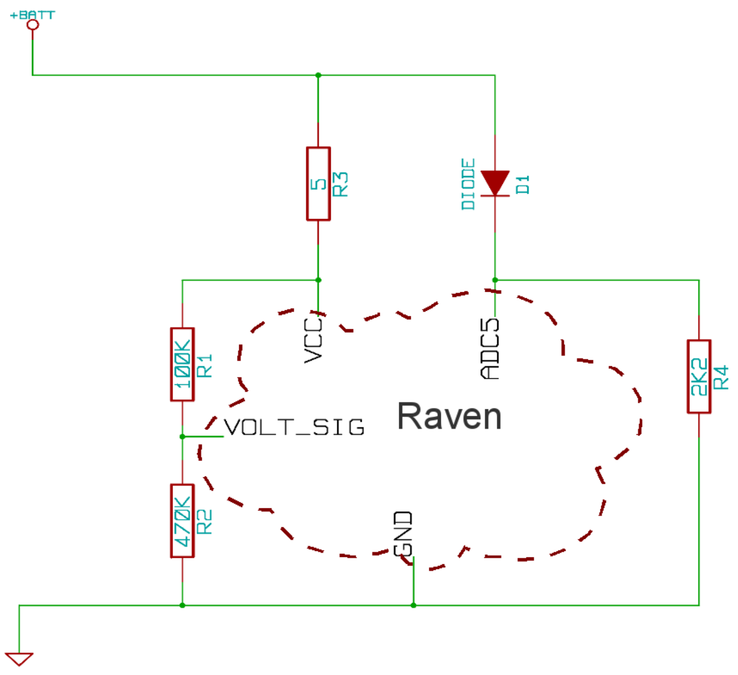


Figure 5.3: *Current Measurement setup for the AVR Raven*

In order to calculate the instantaneous power used by the sensor node, we need to measure the current flowing from the battery source. To this end, we connect a shunt resistor in series with the Raven node. Estimating around $30mA$ current, we choose a 5Ω resistor. This accounts for a voltage drop of about $0.15V$. We would not be able to connect any end of the resistor directly to the ADC, because if we put the resistor in series, at the GND end of the board, then the GND of the board would not be correct and the voltage to be measured would be either 0 or negative. If we connect the resistance at the other end, we would surpass VCC, and thus we would not be able to measure it in the ADC with AVCC reference. Internal reference is no good either, because it is of only $1.1V$. What we do is take the voltage at the + end of the resistor and decrease it by 0.5V with the voltage drop of the diode $D1$. To ensure that $D1$ is opened at all times, we connect it with the GND through the $2.2k\Omega$ resistor $R4$. This ensures the diode is opened with a minimum $1mA$ flowing. The voltage at the cathode of the diode is measured at an ADC input pin with VCC reference.

$$ADC = f \cdot V_{cc_{board}}$$

$$V_{cc_{real}} - 0.5V = f \cdot V_{cc_{board}},$$
$$V_{cc_{real}} = f \cdot V_{cc_{board}} + 0.5V,$$

f is the ratio given by the ADC, in the range $[0, 1]$ with step $\frac{1}{1023}$.

$$V_{cc_{real}} - V_{cc_{board}} = (f - 1) \cdot V_{cc_{board}} + 0.5V$$

$$V_{R_{shunt}} = (f - 1) \cdot V_{cc_{board}} + 0.5V$$

We should also calculate precision:

$$dV_{R_{shunt}} = df \cdot V_{cc_{board}}$$
$$dI \cdot R = \frac{1}{1023} \cdot V_{cc_{board}}$$
$$dI = \frac{V_{cc_{board}}}{1023 \cdot 5},$$

which for $V_{cc_{board}}$ with values between $1.8V$ and $2.3V$, means precision of $0.39mA$ to $0.45mA$, which we consider sufficient for our purposes.

The value of the shunt resistor needs to be chosen correctly, a too small value resistor will have a very small voltage drop, and precision will be low, a too large value resistor will have a big voltage drop, the sensor node's lifetime will be greatly decreased. $0.15V$, as we have chosen, means that the node will have $2.25V$ at full battery (the nodes operates at supply voltages greater than $1.8V$)

The voltage and current measurements will be transferred via the inter-processor USART from the Atmega3290 to the Atmega1284. The time interval in which a Contiki process (associated with a WSN task) is executing is measured, making possible the calculation of the total energy spent during that interval. This information is then relayed after a specified interval from each of the nodes to the scheduler, who can add them up and evaluate the schedule.

5.6 Algorithm

5.6.1 Formalisation of the problem

In this subsection we will try to formalise the problem defined in 5.3. First, we will consider the total energy remaining in a node, W_{m_k}, m_k being the node. This energy can be deduced from the battery voltage and the discharge profile of the type of battery used[‡] . We will use this energy, together with the estimated consumption per second, to deduce the number of seconds that the node can run. The minimum of these times, calculated over each node in the network, will be the network lifetime. The purpose of the scheduling algorithm is to maximize this value.

Because the microcontrollers that we use never enter sleep mode (and their consumption is always very low compared to the transmission/reception power consumption), we will consider the energy they use incorporated into W_{idle}, which can be specific to each node, $W_{idle_{m_k}}$. This value represents the energy consumed by the sensor node during idle mode in one second.

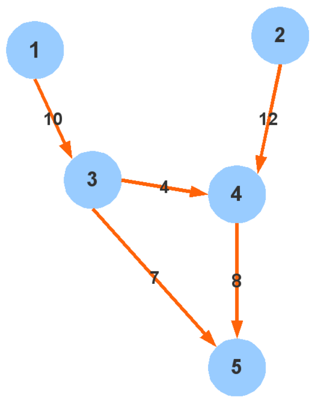


Figure 5.4: *A Directed Acyclic Graph with edges proportional in weight to transmission energy cost.*

We consider the energy wasted in trasmitting/receiving directly proportional to the number of bits in the payload. To model the tasks and their dependencies we use a Directed Acyclic Graph (DAG), in which edges represent data dependencies, their cost being the maximal number of bits transmitted between the tasks. (We consider that a transmission occurs after each period).

Let:

- $T(m_k)$ be the set of tasks allocated to node m_k.

[‡] Note however that the Raven Node can only function at voltage over $1.8V$, so W_{m_k} will be the energy remaining until the battery provides that threshold voltage.

- $W_{idle_{m_k}}$ the idle energy consumed by a node m_k during one second.
- ν_i the frequency (in Hz) of the task v_i.
- $B(e)$, $e \in E$, E the set of edges in the task DAG.
- $W_{tr\,b,m_k}, W_{rcv\,b,m_k}$ the energy cost of transmitting/receiving a payload bit on/from node m_k.
- $M(v)$ is the node to which the task v was assigned.

Then the energy consumed by a node m_k is:

$$W_{idle_{m_k}} + \sum_{i,\,v_i \in T(m_k)} \nu_i \Big(\sum_{j,\,v_j \notin T(m_k)} B(e_{ij}) \cdot W_{tr\,b,m_k} + \sum_{j,\,v_j \notin T(m_k)} B(e_{ji}) \cdot W_{rcv\,b,m_k} \Big) \tag{5.1}$$

Thus the network lifetime is:

$$\max_{k,\,m_k \in M} \frac{W_{m_k}}{W_{idle_{m_k}} + \sum_{i,\,v_i \in T(m_k)} \nu_i \Big(\sum_{j,\,v_j \notin T(m_k)} B(e_{ij}) \cdot W_{tr} + \sum_{j,\,v_j \notin T(m_k)} B(e_{ji}) \cdot W_{rcv} \Big)} \tag{5.2}$$

Taking into account that $W_{idle_{m_k}}$ is almost the same on all nodes, and presuming that we have the same amount of energy available from the batteries, the important part remains the energy consumed in radio transmission. Thus, to maximize network lifetime, a general goal would be to minimize this consumption. We do not consider tasks that need to be run on all capable nodes, we will address this as a restriction in the algorithm.

We have accounted for both energy while transmitting and while receiving. Even if they are not exactly the same, their sum should be uniform over each bit transmitted, so we can say that the energy consumed by the network in radio communication, W_{radio} is

$$W_{radio} = \sum_{i,j,M(v_i) \neq M(v_j)} B(e_{ij}) * (W_{tr\,b,m_k} + W_{rcv\,b,m_k}) = \sum_{i,j,M(v_i) \neq M(v_j)} B(e_{ij}) * K \tag{5.3}$$

We have reduced the scheduling problem to a known graph-problem, for which a polynomial algorithm has been found in 1988. It is called the min k-cut problem. If we imagine in our setup the assignment of tasks to nodes, and ascertain that no task is duplicated among nodes (we can enforce that easily by duplicating in advance tasks that have to be run on all nodes), then the scheduling is in fact a partition of the node sets $\{C_1, C_2, ..., C_k\}$, each resulting set containing the tasks that have to run on that node. In graph theory, this is called a *k-cut*.

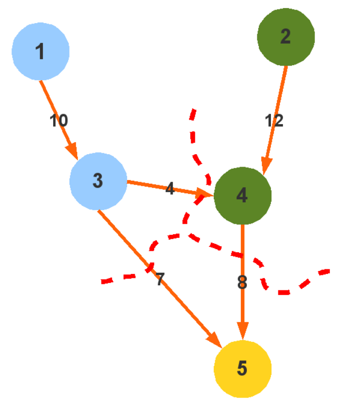


Figure 5.5: *A Directed Acyclic Graph with a minimal 3-way cut. Note that even though on this particular graph the cut is representable in 2D, this is not always true.*

5.6.2 Minimal K-Cut

Given a graph $G = (V, E)$ and a weight function $w : E \to \mathbb{N}$ and an integer $k \in [2..|V|)$ the k-cut is a partition of V into k disjoint sets $F = \{C_1, C_2, ..., C_n\}$ and its measure is the sum of the weight of the edges between the dijoint sets

$$\sum_{i=1}^{k-1} \sum_{j=i+1}^{k} \sum_{v_1 \in C_i v_2 \in C_j} w(v_1, v_2)$$

or otherwise written

$$\sum_{i,j,\, v_i, v_j \text{ in different sets}} w(v_1, v_2)$$

The minimal k-cut algorithm is described in Listing 5.1

Listing 5.1: Min K-cut algorithm

```
KCut(V,k)
  if k is even
    k' = k - 2
  else
    k' = k - 1
  S ← the set of subsets of k' elements from V
  T ← the set of subsets of k-1 elements from V

  Find $s \in S, t \in T$ such that W(cut(s,t)) = min
  /* cut(s,t) splits V into s' and t'
  /* Find the minimal cut(s,t) with maximal source set */
  return s' ⋃ KCut(V−s',k-1)
```

5.6.3 The maximal source minimal s-t cut

The (s, t) cut, needed by the min K-Cut algorithm, is a partitioning of the vertices of a flow graph such that the source is in S and the sink is in T. The min K-Cut algorithm needs a version where the source and sink are a set of nodes, not just one. To do this, we can collapse the nodes in the sink set into a supernode. Edges on the interior of the supernode do not count for the search of the minimal s-t cut, only those on its exterior. We then proceed to solve the maximum-flow problem on the graph, interpreting weights as flow capacities. Using the residual graph (graph with edges that have weight the capacity - flux passing at one time), we can start from the sink and expand until we hit 0 residual capacity edges. The nodes found will be the smallest sink set of a minimal s-t cut, the source set being the rest of the nodes.

$$r(i,j) = c(i,j) - f(i,j)$$

Listing 5.2: Maximal source minimal s-t cut

```
mstCut(S,T)
  /* replace sink and source set by supernodes */
  $V' = V \bigcup \{s,t\} - (S \bigcup T)$
  modify the edges external to supernodes
  $E' = \{e_{ij} | i,j \notin S,T\} \bigcup \{e_{si} | e_{ji} \in E \, and \, j \in S\} \bigcup$
  $\{e_{ti} | e_{ji} \in E \, and \, j \in T\} \bigcup \{e_{st} | e_{ij}, i \in S, j \in T\}$
  $F_{ij} = 0, \forall i,j \in V$
  while 1
    find path $p$ from $s$ to $t$ in residual graph
    $m \leftarrow$ minimum residual capacity on path $p$
    for each edge $e_{ij}$ on path $p$
      $F_{ij} \leftarrow F_{ij} + m$
      $F_{ji} \leftarrow F_{ji} - m$

  $A \leftarrow$ set of nodes reachable by BFS from $t$
  $B \leftarrow V - A$
  return $(A,B)$
```

5.6.4 Adaptation of K-Cut

When reducing the scheduling problem to K-Cut, some constraints were ignored that now must satisfied. Since every step of the alghoritm is partitioning the node set in half, one being final and one remaining to be further partitioned, we can say that each step represents scheduling tasks on a single node. Contraints must be inserted in each step, relevant to the node whose tasks are being scheduled.

In the pseudocode of the algorithm, finding the source set S is what must be modified to satisfy constraints.

We have several constraints that must be added to the algorithm:

- Some tasks can only run on compatible nodes
- Some tasks have to run on all capable nodes (e.g. sensing tasks)

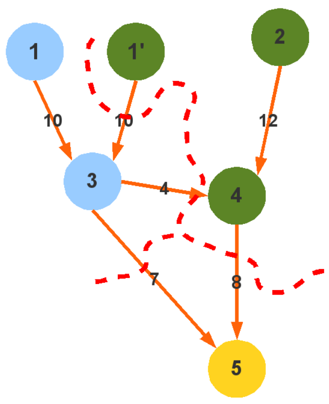


Figure 5.6: *A directed graph with a task that has to be duplicated on all capable nodes*

To enforce the first constraint we have to include only compatible tasks (tasks v such that $NA(v, m) = 1$ for the current node m) in the source set, as well as filter the cuts in which the first resulting set contains incompatible tasks. For the second constraint, we will include the tasks that have to be duplicated in the sink set of the cut (so that they will be available for the next step of the algorithm), then in the end we add those tasks to those obtained in the minimal (s, t) cut.

Listing 5.3: Adapted min K-Cut

```
MKCut(V,k,$m_i$)
  if k is
    k' = k - 2
  else
    k' = k − 1
  MT ← {tasks that have multiplicity}
  V' ← V - MT
  S ← {the set of subsets of k' elements from V'}
  T ← {the set of subsets of k-1 elements from V'}$\bigcup$ MT

  Find $s \in S, t \in T$ such that W(cut(s,t)) = min
  /* cut(s,t) splits V into s' and t'
```

```
    /* Find the minimal cut(s,t) with maximal source set */
    $T(m_i) = \{s'\} \bigcup \{v_j | v_j \in$ MT, $NA(v_j, m_i) = 1\}$
    return $T(m_i) \bigcup$ KCut(V-s',k-1,$m_{i+1}$)
```

Assigning the tasks for each node at each step gives great versatility in constraints management for the algorithm to better model and solve the problem. For instance, battery status has no role yet, but it is easy to add: We set a threshold for the battery/tasks ratio, if the current node crosses that threshold we will settle on another solution, not necessarily with the same min k-cut, but with less strain on the node.

5.6.5 Complexity

The complexity of the min k-cut with the algorithm we described is:

$$T(n) = O(n^{k'+k+1}) \cdot O(n^3) + T(n-1),$$

where $O(n^3)$ is the bound for the minimum (s,t)-cut algorithm.
For k even, we have:

$$O(n^{2k-3}(n^3 + n^{2k-4}(n^3 + ...n^4(n^3 + n^3)))))) = O(n^{k^2})$$

A precise evaluation gives, as found by [GH88]:

$$O(n^{k^2-3k/2+2}),\ k\,even$$
$$O(n^{k^2-3k/2+5/2}),\ k\,odd$$

5.7 Approximation Alternative

5.7.1 Gomory-Hu trees

We will construct then *Gomory-Hu* trees (or cut trees). From those we will deduce the cut with minimal weigth and maximal source set needed for our scheduling algorithm.

Let $G = (V, E)$ be a graph and $c : E \to \mathbb{R}$ a cost function for its edges. Let T be a tree with the node set V, and c' a cost function for edges in T. Then the T is a Gomory-Hu tree for G if:

- $\forall\, s, t \in V$, the cost of the minimum $s - t$ cut in G coincides with the cost of the minimum $s - t$ cut in T.
- $\forall$ edges $[s, t] \in T$, $c'(x, y)$ is the cost of the minimum $s - t$ cut in G.

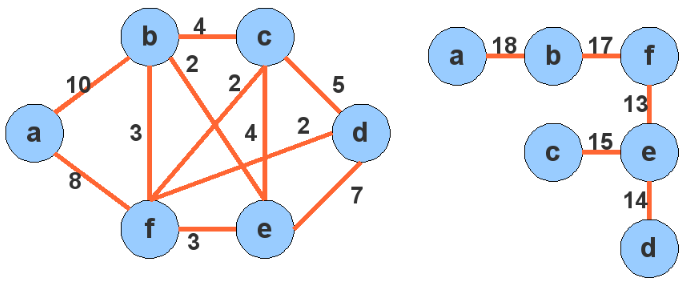


Figure 5.7: *A graph (left) and its Gomory-Hu tree (right)*

5.7.2 Approximation Algorithm

The algorithm is based on a theorem from [Jac06]:

Removing l cuts associated with l edges in the Gomory-Hu tree for G results in a new graph G' with at least l+1 connected components.

The idea behind the algorithm is to build the Gomory-Hu tree of graph G, named T. We will take the lightest $k-1$ cuts from the $n-1$ cuts associated with the graph G, leaving at least k components.

5.8 Integration in SENSEI

In the SENSEI project, the scheduling algorithm presented here appears under *Check - Scheduling for Task-based WS&ANs*. It is implemented as a service available through multiple interfaces, some standard SENSEI interfaces and some non-standard.

RAI is a standard SENSEI interface (stands for Resource Access Interface), that will enable access to the different functionalities of the WS&AN island:

- *Collecting Data* - RAI.get("parameters") would gather sensor data.
- *Starting/Stopping tasks* - RAI.set("start") or RAI.set("stop") would start or stop tasks in a WS&AN island. This is tied to the dynamic scheduling functionality.
- *Sending Output to Multiple Sinks* - This is automatically taken care of if a task has several succesors in a data-dependency graph of the application.
- *Tasks: List,Status,Logs* - Allows for monitoring the island, generating statistics or investigation of failures.
- *Multi-hop Scheduling* - Extension of the algorithm for multi-hop situations.

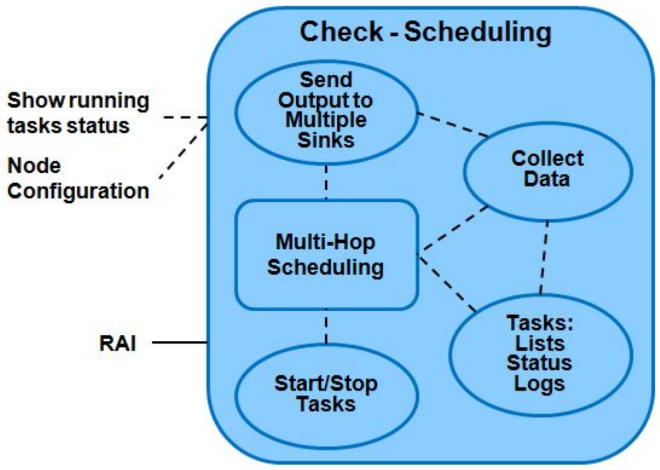


Figure 5.8: *Check - Scheduling as a SENSEI Building Block*

As shown in Figure 5.9, Check Scheduling has a couple of dependencies in the SENSEI project, one is from the Connectivity domain, multi-hop routing, the other is related to Management of WSNs, Reprogramming, Monitoring and Self-healing. In the figure, blue represents middleware (The Check-Scheduling Algorithm is considered to be a middleware component), pink stands for management and yellow for connectivity.

Multi-hop routing is needed to extend the algorithm to a network with graph topology. Reprogramming Over-the-Air (OTA) is needed for faster development, Monitoring is related to the data that Check - Scheduling exposes through the RAI interface and the Self-healing would be one of the services that uses the Check-scheduler, reconfiguring the tasks around the WS&AN island to replace node failures in the WSN network.

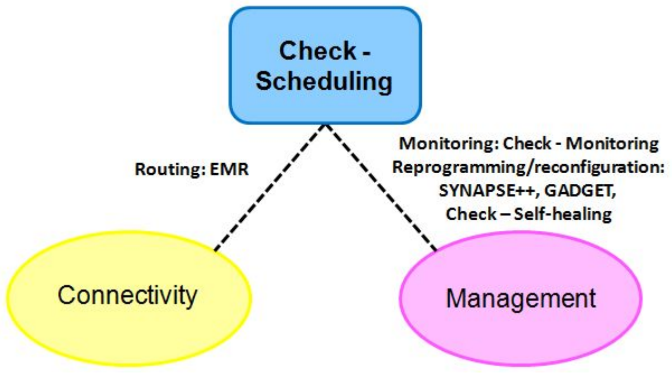


Figure 5.9: *Check - Scheduling Dependencies*

Chapter 6

Dynamic Scheduling

The dynamic version of the algorithm can burrow heavily from the static one. The need for dynamics arises from the interface with outside components that will control some aspects of the wireless application running. Even if the tasks are already present on the wireless nodes as chunks of code, starting and stopping them could be arbitrarily decided from outside the network. In this case, the speed with which the new task is assigned or the rescheduling takes place gives the responsiveness of the network, which can be crucial in some respects.

We will propose two ways of assigning new tasks, one is a greedy approach and the other is based on an update of the model described and calculated with the static scheduling algorithm. The former will be a lot faster, but will differ from the optimum solution.

6.1 On-the-fly Dynamic Scheduling

The dynamic problem is inherently that of starting a new task. In the on-the-fly method, stopping tasks holds no rescheduling consequences. Even with using a Greedy approach, the algorithm can obtain good results is a good minimization function is found. The purpose is still to maintain an extended network lifetime. If a node has to receive a new task, it could be the one with the most network lifetime:

$$M(v_{new}) = \max_{k,\, m_k \in M} Lifetime(m_k) \tag{6.1}$$

$$M(v_{new}) = \max_{k,\, m_k \in M} \frac{W_{m_k}}{W_{idle_{m_k}} + W_{communication_{m_k}}} \tag{6.2}$$

However, this would not be the best approach because the node with the longest remaining lifetime might not be compatible, or dependencies of the new task could be quite costly. The minimization function must take into account the impact on the node's communication energy consumption.

$$f(v_{new}, m_k) = \min_{k, m_k \in M} \frac{\frac{1}{L(m_k)} + \alpha \cdot (L'(m_k) - L(m_k))}{NA(v_{new}, m_k)}, \tag{6.3}$$

where:

- $L(m_k)$ - The lifetime of node m_k.
- $L'(m_k, v_{new})$ - The lifetime of node m_k after v_{new} is assigned.
- $NA(v_{new}, m_k)$ - The affinity of the new task to the node m_k.
- α - A balancing coefficient between keeping the greatest network lifetime (picking the node with the most battery) and network-level energy saving (choosing a node for our new task such that extra communication is minimized).

6.2 Recalculating the static schedule

Another variant of dynamic scheduling is based on the approximation algorithm. Gomory-Hu trees can be recalculated quite fast, as the s-t cuts do not change much with the addition of an extra edge (the atomic operation is considered here to be adding an edge, adding a node to a DAG does not change anything if it is not connected). Some of the computations can be retained so that calculations can resume right where the new edge was inserted and finish quite quickly, even faster than the normal approximation algorithm.

Chapter 7

Implementation

This chapter presents some aspects of the implementation worth mentioning. Some parts of the tasks running on the wireless sensors, how the algorithm for the scheduler is implemented.

7.1 Tasks

This section presents some archetypes of Contiki tasks, most of the tasks needed for a WSN application as described in this study can be implemented from this basis. We include parts of the metatask, one of the tasks that monitor the node and a framework for a sensing task.

7.1.1 Metatask

The meta-task is the process on each sensor node responsible for task management. It is implemented as a server, any incoming connection is treated as a sequence of commands to start/stop or modify the tasks. Nodes begin with a list of implemented tasks predefined (in agreement with their capabilities). The meta-task is automatically started on each reset and maintains a list of the other tasks and their status.

Listing 7.1: Task server snippet

```
PROCESS_THREAD(task_server_process, ev, data)
{
    PROCESS_BEGIN();

    list_init(task_list);

```

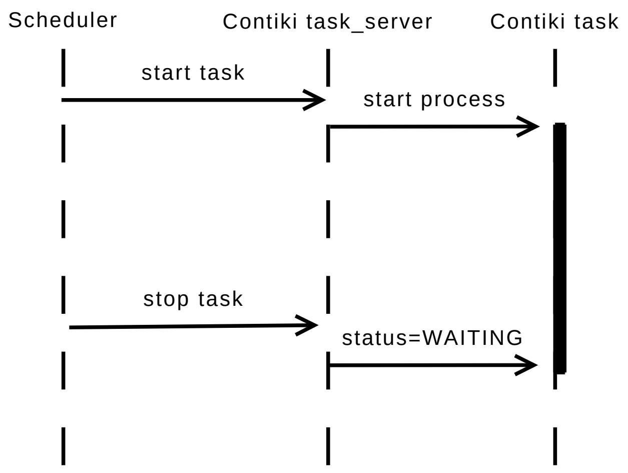


Figure 7.1: *The exchange of messages while starting/stopping tasks*

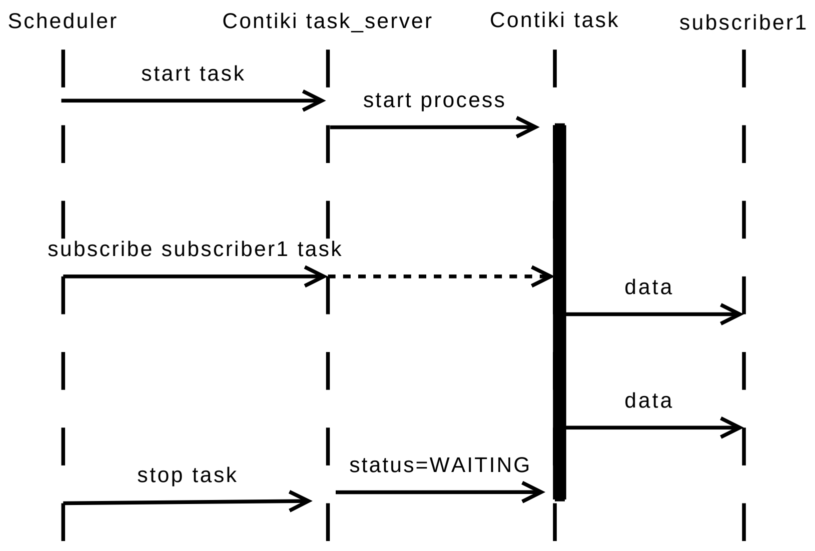


Figure 7.2: *The exchange of message during subscription of a task on another node to the output of a task on the current node*

```
    list_add(task_list,&el_monitor_process);
    list_add(task_list,&el_delay_process);
    list_add(task_list,&el_temperature_sensing);

    tcp_listen(HTONS(1010));

    while(1)
    {
        PROCESS_WAIT_EVENT_UNTIL(ev == tcpip_event);

        if(uip_connected())
        {
            PSOCK_INIT(&ps, buffer, sizeof(buffer));

            while(!(uip_aborted() || uip_closed()
                || uip_timedout()))
            {
                PROCESS_WAIT_EVENT_UNTIL
                        (ev == tcpip_event);
                handle_connection(&ps);
            }
        }
    }
    PROCESS_END();
}
```

7.1.2 Load monitor process

The load monitor is the tool discussed in 5.4.3, which helps determine whether a node can take extra processes or not. Generally we believe that a collaborative system will have a maximum number of tasks, highly dependendant on the programming of each one, of their frequencies of communication, activity and so on. This metric might also be called a responsiveness metric, the value is the time between schedules.

Listing 7.2: Load monitor process

```
PROCESS_THREAD(monitor_process, ev, data)
{
    PROCESS_BEGIN();
    while(1)
    {
        t1 = clock_time();
        PROCESS_PAUSE();
        dif = clock_time() - t1;
    }

    PROCESS_END();
}
```

7.1.3 Sensing Task

Listing 7.3: Sensing taks (generic) snippet

```
\small
PROCESS_THREAD(sensing_process, ev, data)
{
    PROCESS_BEGIN();

    while(1)
    {
        tatic struct etimer et;


        etimer_set(&et, CLOCK_SECOND * 10);
        PROCESS_WAIT_EVENT();

        if(etimer_expired(&et))
        {
            while(tc.forwarding)
            {
                PROCESS_PAUSE();
            }

            for(i = 0; i < el_sensing_process.nrsub; i++)
            {
                /* form the IPv6 address */
                uip_ip6addr(paddr,
                    el_sensing_process.sub[i][0],
                    el_sensing_process.sub[i][1],
                    el_sensing_process.sub[i][2],
                    el_sensing_process.sub[i][3],
                    el_sensing_process.sub[i][4],
                    el_sensing_process.sub[i][5],
                    el_sensing_process.sub[i][6],
                    el_sensing_process.sub[i][7]);
                tcp_connect(paddr,HTONS(4242),NULL);

                PROCESS_WAIT_EVENT_UNTIL
                    (ev ==tcpip.event);

                if (uip_connected())
```

```
                    {
                        PSOCK_INIT(
                          &el_sensing_process.protosock,
                          buffer,sizeof(buffer));

                        do
                        {
                            handle_send(&ps2);
                        }while (!(uip_close() ||
                            uip_aborted() ||
                            uip_timedout()));
                    }
                }
            }
        }
        PROCESS_END();
}
```

7.2 Scheduler

The scheduler itself is currently written in Python. Python was chosen for its ease of development and higher-level primitive data types that are well suited for more complex algorithms such as this one.

The scheduler uses pygraph as a framework for basic graph operations (adding edges, removing nodes, etc.), and graphviz for visualising data.

Listing 7.4: Main K-Cut algorithm

```
def kcut(graph,k):
    kp= k - 2 + (k % 2)
    if kp < 1:
        kp = 1
    if k == 1:
        return [graph.nodes()]

    #tasks that have multiplicity
    mt = [e for e in graph.nodes()
        if tasks[e]['multiplicity']>1]

    st = affine_possibilities(
        list(set(graph.nodes()).difference(set(mt))),
        kp,
        len(battery)-k)
    tt = possibilities(graph.nodes(),k-1)

    mincut = []
    for s in st:
        for t in tt:
            if len(set(s).intersection(set(t))) == 0:
                res = stcut(graph,s,t)
                if len(mincut)==0:
                    mincut = res
                else:
                    if res[2] < mincut[2]:
                        mincut = res
    for old in mincut[0]:
        graph.del_node(old)
    return [mincut[0]+mt] + kcut(graph,k-1)
```

Chapter 8

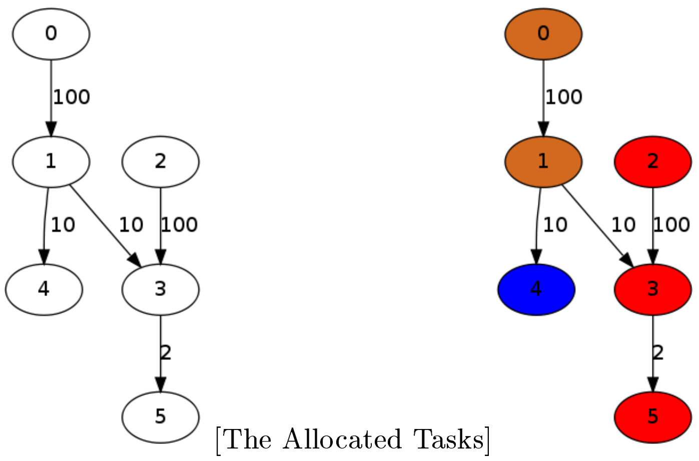
Scenario(s)

We will propose a simple application to begin with and see how the scheduler behaves, then we will show several schedules for random sets of tasks.

The scenario is about a factory in which there are a lot of shortcircuits happening. Each shortcircuit has a 10% chance of igniting a fire, and each shortcircuit has 10% chance of occuring in any large enough interval of time. We would like to have a level-based alarm system, where the network detects a shortcircuit and only then checks for immediate increases in temperature. This application can be obviously extended in multiple ways (more wireless nodes, more types of sensors). The factory uses 220V dual-phase AC.

The system proposed has one Sparrow Power node and two AVR Raven™ boards, and is broken down into 6 tasks.

The result is that the voltage related tasks are assigned to the Sparrow Power,



[The Task DAG] [The Allocated Tasks]

Figure 8.1: The scenario

while the other tasks are split in two: Shortcircuit alarm is delegated to one Raven, while the other checks for increases in temperature and has a fire alert.

Task name	Task no.	Frequency	Affinity
Voltage sampling	0	1	Sparrow Power
Voltage decrease detection	1	1	Sparrow Power
Temperature sensing	2	1	AVR Raven
Fire detection	3	1	AVR Raven
Shortcircuit alarm	4	1	AVR Raven
Fire alarmn	5	1	AVR Raven

Table 8.1: Tasks of scenario

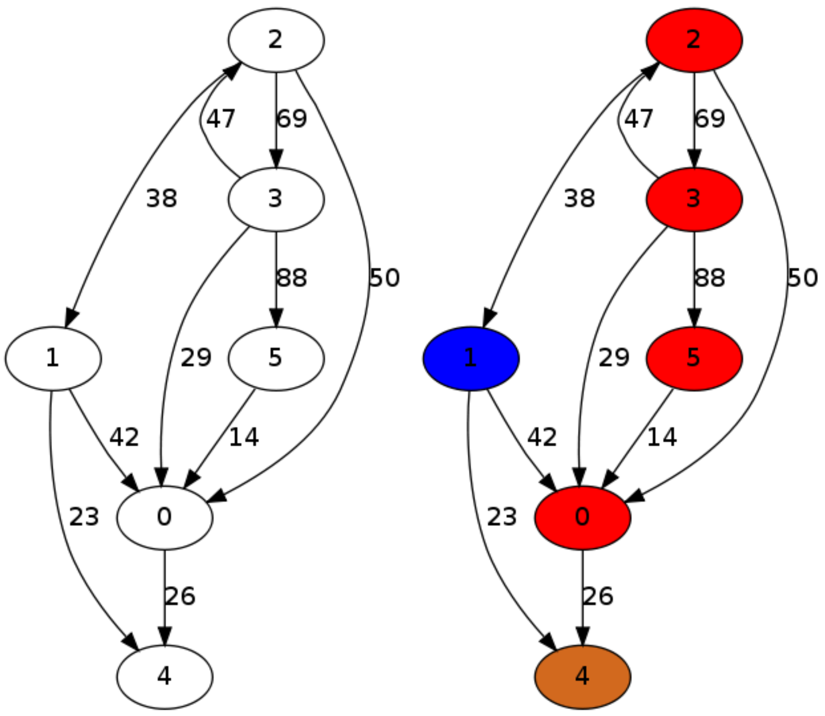


Figure 8.2: A random scenario

Chapter 9

Conclusions

This study has shown that scheduling on Wireless Sensor Networks can be approached as a graph-cut problem with success, because graph algorithms are easy to implement and at the same time powerful and versatile. Its versatility was clear in 5.6.4, where new constraints related to energy efficiency were easily integrated into the algorithm.

The main contribution of this work was defining the problem of task scheduling as a more accurate reflection of reality, together with considering this a clustering / minimal cost partition (min k-cut) problem. The model was not destined to be simulated, it was based on real hardware platforms and inputs of the scheduler are consistent with the real state of the network in a moment of time. This makes the implementation of the algorithm easy and consistent with the theory.

Even though the asymptotical complexity is quite large in the case of this algorithm, the static case can be expected to appear more often, with time for scheduling not being a constraint. The main constraint in wireless sensor networks is energy, and scheduling the tasks in a network can finish before the network is enabled. Aproximation methods exist and their running times are a lot smaller for at most twice the energy spent.

With rising interests in smart interfaces, services and self-organizing networks, dynamic scheduling is increasing in importance. We've outweighed two directions of optimization for greedy schedulers, one to maximize network lifetime and one to minimize energy spent communicating (which are not the same criteria).

In the final part we have shown how an application for wireless sensor networks can be broken down into tasks and how to represent their hardware requirements (we used the term *node affinity*. Tasks can be of a few varieties, sensing tasks (personalized for each sensor or each node - hardware specific), event detection tasks, actuating tasks (the reverse of sensing tasks) and generic processing tasks. Along with these we have added *metatasks* which aid in the process of scheduling and network status monitoring.

9.1 Outlook

There are two issues that could not be discussed as in depth as it was necessary. One is the influence of multi-hop communication on energy consumption, that we have not accounted for and the other is the number of tasks that is optimum for a given topology and data dependencies.

These issues are closely related and could not be explored due to lack of multi-hop routing in our software environment, which will be available some time after this study is finished.

Multi-hop communication would significantly alter the formalisation of the problem. Energy would no longer be spent only by the transmitter and the receiver, but by any node that forwards the packets between them, with double the energy (receive / transmit). The scheduler would need to be fully-aware of the topology and calculate the cost of each data dependency proportional to the number of hops needed to reach the destination.

The second issue borders self-organisation ideas for wireless sensor networks, but instead of choosing a local leader, the scheduler chooses how many times to assign a task. This would lead to the tasks that gather data from other sensors and process them to be closer to the source of the data then before.

There is a method to implement this last idea within a single-hop network, but it would not be as effective as in a multi-hop one. The idea resembles the one for tasks that have to run on all capable nodes. Because the algorithms makes steps of 2-cuts, we will discuss that case: Let us suppose that we have already obtained the minimum s,t-cut, with the tasks with variable number of instances all in the sink set. We would assign a copy to the source set only if the sum of the edges going to and from nodes outside the source set are smaller than the sum of edges going to and from the nodes inside the source set.

Bibliography

[Atm] Atmel. Rzraven hardware user's guide. [cited at p. -]

[Cal04] Edgar H. Callaway. *Wireless Sensor Networks: Architectures and Protocols.* CRC Press, 2004. [cited at p. -]

[CK03] C. Chong and S.P. Kumar. In *Sensor networks: Evolution, opportunities, and challenges*, 2003. [cited at p. 3]

[DPdR$^+$05] Flvia Coimbra Delicato, Fbio Protti, Jos Ferreira de Rezende, Luiz F. Rust da Costa Carmo, and Luci Pirmez. Application-driven node management in multihop wireless sensor networks. In Pascal Lorenz and Petre Dini, editors, *ICN (1)*, volume 3420 of *Lecture Notes in Computer Science*, pages 569–576. Springer, 2005. [cited at p. 13, 32]

[ea07] Roberto Verdone et al. *Wireless Sensor and Actuator Networks: Technologies, Analysis and Design.* Academic Press, 1st edition, 2007. [cited at p. -]

[GH88] Olivier Goldschmidt and Dorit S. Hochbaum. Polynomial algorithm for the k-cut problem. 1988. [cited at p. 42]

[HJ05] Tarek Hagras and Jan Janecek. A high performance, low complexity algorithm for compile-time task scheduling in heterogeneous systems. *Parallel Computing*, 31(7):653–670, 2005. [cited at p. 10]

[Jac06] Brian Jacokes. Lecture notes on multiway cuts and k-cuts, July 2006. [cited at p. 43]

[KD06] Snehal Kamalapur and Neeta Deshpande. Efficient cpu scheduling: A genetic algorithm based approach. *Ad Hoc and Ubiquitous Computing*, pages 206 – 207, December 2006. [cited at p. -]

[Li08] Xiang-Yang Li. *Wireless Ad Hoc and Sensors Networks: Theory and Applications.* Cambridge University Press, 2008. [cited at p. -]

[NGT99] Andrew Goldberg Nec, Andrew V. Goldberg, and Kostas Tsioutsiouliklis. Cut tree algorithms. In *In Symposium on Discrete Algorithms*, pages 376–385, 1999. [cited at p. -]

[PB05] C. Prehofer and C. Bettstetter. Self-organization in communication networks: principles and design paradigms. *IEEE Communications Magazine*, 43(7):78–85, July 2005. [cited at p. 4]

[TEÖ07] Yuan Tian, Eylem Ekici, and Füsun Özgüner. Energy-constrained task mapping and scheduling in wireless sensor networks. 2007. [cited at p. 7]

[YJK07] William M Iversen Yunseop (James) Kim, Robert G Evans. The future of intelligent agriculture. *Resource*, October 2007. [cited at p. 17]

[zig] Zigbit wireless modules datasheet. [cited at p. -]

Appendix A

Contiki API

A.1 Process macros

- `PROCESS_THREAD (name, ev, data )` - Define the body of a process. This macro is used to define the body (protothread) of a process. The process is called whenever an event occurs in the system, A process always starts with the PROCESS_BEGIN() macro and end with the PROCESS_END() macro.
- `PROCESS_BEGIN ()` - Define the beginning of a process.
- `PROCESS_END ()` - Define the end of a process.
- `PROCESS_YIELD ()` - Yields the currently running process
- `PROCESS_WAIT_EVENT_UNTIL (c)` - Wait for an event to be posted to the process, with an extra condition. This macro is very similar to PROCESS_WAIT_EVENT() in that it blocks the currently running process until the process receives an event. But PROCESS_WAIT_EVENT_UNTIL() takes an extra condition which must be true for the process to continue.
- `PROCESS_PAUSE` - Yield the process for a short while. This macro yields the currently running process for a short while, thus letting other processes run before the process continues.

A.2 uIP functions

- `PSOCK_INIT (psock, buffer, buffersize)` - Initializes a proto-socket. This macro initializes a protosocket and must be called before the protosocket is used. The initialization also specifies the input buffer for the protosocket.

- `PSOCK_SEND (psock, data, datalen)` - Send data. This macro sends data over a protosocket. The protosocket protothread blocks until all data has been sent and is known to have been received by the remote end of the TCP connection.
- `PSOCK_READBUF (psock)` - Read data until the buffer is full. This macro will block waiting for data and read the data into the input buffer specified with the call to PSOCK_INIT(). Data is read until the buffer is full..
- `CCIF process_event_t tcpip_event` - The uIP event. This event is posted to a process whenever a uIP event has occurred.
- `CCIF void tcp_listen (u16_t port)` - Open a TCP port. This function opens a TCP port for listening. When a TCP connection request occurs for the port, the process will be sent a tcpip_event with the new connection request.
- `struct uip_conn *tcp_connect(uipipaddr_t *ripaddr,u16 port, void *appstate)` - This function opens a TCP connection to the specified port at the host specified with an IP address. Additionally, an opaque pointer can be attached to the connection. This pointer will be sent together with uIP events to the process.
- `uip_connected()` - Has the connection just been connected?
- `uip_closed()` - Has the connection been closed by the other end?
- `uip_aborted()` - Has the connection been aborted by the other end?
- `uip_timedout()` - Has the connection timed out?
- `uip_newdata()` - Is new incoming data available?
- `uip_close()` - Close the current connection.

Appendix B

Node capabilities

Task	AVR Raven™	Sparrow	Sparrow Power
Temperature sensing	✔	✔	
Humidity sensing		✔	
Voltage & Current sensing			✔
Event detection	✔	✔	✔
Alarm beep	✔		
LED signal	✔	✔	

Table B.1: Node capabilities

List of Figures

List of Tables

Listings